MASTER EN SCIENCES INFORMATIQUES

Analyse d'exécutables x86 par transformation sous forme de clauses de Horn

Devos, Axel

Award date:
2017

Awarding institution:
Universite de Namur

Université de Namur
Faculté d'informatique
Année académique 2016-2017

Analyse d'exécutables x86 par transformation sous forme de clauses de Horn

Devos Axel

Maîtres de stage : Mesnard Frédéric, Payet Étienne

Promoteur : (Signature pour approbation du dépôt REE art. 40)
Vanhoof Wim

Co-promoteur : /

Confidentialité du mémoire

Mémoire présenté en vue de l'obtention du grade de
Master en Sciences Informatiques.

Résumé

Intel et AMD sont les leaders mondiaux quant à la production de processeurs, à la fois destinés aux ordinateurs personnels qu'aux serveurs industriels. Ils représentent à eux seuls plus de 90% de la part du marché mondial.

De ce constat, il en va de soi que la majorité des programmes et logiciels implémentés en langage de haut niveau soient compilés en vue de s'exécuter sur des processeurs Intel et AMD. Comme ces processeurs partagent la même architecture (x86), il serait donc intéressant de pouvoir étudier et réaliser diverses analyses de programmes sur ces programmes exécutables mais paradoxalement, ceci reste assez complexe et très peu d'études existent à ce sujet.

Cela s'explique en grande partie par le fait que ces processeurs se doivent d'être rétrocompatibles entre eux. En effet, un programme s'exécutant sur un processeur Intel Pentium 4 (2000) doit également pouvoir s'exécuter sur un Intel Core i7 (2008). Cela signifie que le jeu d'instructions qu'exécutent les processeurs actuels doit contenir les instructions utilisées par les processeurs précédents. Cette architecture de jeu d'instructions est le *code machine x86* et les programmes exécutables sur des processeurs compatibles avec ce jeu d'instructions sont dits *exécutables x86*.

L'intérêt de ce mémoire est de rendre possible ou de simplifier l'application d'analyses de programmes sur des programmes exécutables x86. L'approche que nous proposons est d'élaborer et de concevoir une méthode de traduction en clauses de Horn et plus précisément en programmation logique par contraintes. L'avantage des clauses de Horn est qu'elles bénéficient d'un large ensemble d'analyses de programmes réalisées à leur sujet et qu'elles s'y prêtent particulièrement bien. La technique mise au point et expliquée dans ce mémoire se base sur une traduction séquentielle des instructions du code machine x86 en leur équivalent en clauses de Horn.

Avant-propos

Ce mémoire est présenté en vue de l'obtention du grade de Master 120 en Sciences Informatiques à l'Université de Namur. Afin de fixer le cadre de ce dernier, l'analyse d'exécutables x86 par transformation sous forme de clauses de Horn est un sujet de recherche sur lequel j'ai du travailler tout au long d'un stage universitaire effectué à l'Université de la Réunion, et plus précisément au Laboratoire d'Informatique et de Mathématiques.

Durant ce stage, l'objectif principal fut la réalisation d'un outil capable de transformer un programme exécutable x86 binaire en un ensemble de clauses de Horn afin d'en faciliter les analyses de programmes. Ce mémoire est donc une suite directe de ce stage et présente donc mes recherches, mes analyses, mes résultats ainsi que les problèmes auxquels j'ai été confrontés.

Je tiens, dans un premier temps, à remercier l'Université de Namur et l'Université de la Réunion pour m'avoir offert la possibilité d'effectuer ce stage qui m'a permis d'approfondir grandement mes connaissances sur le monde de l'informatique à bas niveau.

Ensuite, je remercie tout particulièrement mon promoteur, Monsieur Wim Vanhoof, pour le suivi régulier apporté tout au long de mes recherches, pour l'aide fournie concernant l'approche de traduction que j'ai adoptée ainsi que pour ses conseils judicieux et remarques concernant la rédaction de ce mémoire.

Enfin, je remercie également mes maîtres de stage, Monsieur Frédéric Mesnard et Monsieur Étienne Payet, pour leur suivi et leurs connaissances apportées tout au long de mon stage afin de me guider dans les choix à faire et dans les directions à prendre.

Table des matières

Table des figures

Liste des tableaux

Chapitre 1

Introduction

Ce mémoire a été réalisé dans le cadre d'un projet de recherche qui étudie la transformation de plusieurs types de programmes en clauses de Horn afin de simplifier les analyses de programmes sur ceux-ci. Dans ce mémoire nous étudierons la transformation de programmes exécutables x86 sous forme de clauses de Horn.

1.1 Les processeurs x86

Par définition, les processeurs de la famille x86 sont les processeurs étant compatibles avec le jeu d'instructions utilisé par le premier processeur x86 : l'Intel 8086. Ce dernier, créé par Intel en 1978, fut le premier processeur utilisant le jeu d'instructions x86. Il était doté d'un espace d'adressage linéaire de 16 bits, ainsi que d'un système de segmentation basé sur 20 bits permettant donc de bénéficier d'un espace mémoire plus grand que 64 kibioctets[5].

Le code machine x86 est donc le langage natif des processeurs de la famille x86, il manipule des registres et des accès (lecture et écriture) en mémoire. La traduction binaire de chaque instruction x86 est obtenue via une table de traduction contenant le code opération (*opcode*) de chaque instruction.

L'Intel 8086 fut le premier vrai succès commercial de la société Intel et celle-ci continua donc a faire évoluer et à maintenir ses processeurs x86 jusqu'au jour d'aujourd'hui. Face à ce succès, d'autres constructeurs, comme AMD, Cyrix et Via, ont fait le choix d'intégrer le code machine x86 dans leurs processeurs. Lors de l'évolution du code machine ou du processeur lui-même, les concepteurs se retrouvent confrontés à l'aspect de rétrocompatibilité. En effet, un processeur A est dit compatible avec un processeur B si l'ensemble des instructions de ce premier est également exécutable sur ce dernier. La rétrocompatibilité entre processeur permet donc à un programme compatible avec un processeur d'ancienne génération de fonctionner également sur un processeur de nouvelle génération. En termes plus techniques, cela signifie que l'ensemble d'instructions compatible avec un ancien processeur sera également compatible avec un nouveau processeur.

Au fur et à mesure des années, les processeurs x86 ont évolué, au même titre que les différentes technologies utilisées. A l'heure actuelle (2016-2017), nous sommes actuellement à la quinzième génération des processeurs x86 (Intel KabyLake, AMD Zen) et la différence entre la technique (64 bits d'espace d'adressage) ainsi que la technologie (*MMX* [5],...) d'aujourd'hui et celles des années 90 est telle que maintenir la rétrocompatibilité avec les anciens processeurs est devenu un réel problème pour le code machine x86. Celui-ci se retrouve encombré par d'anciennes instructions qui ne sont plus optimisées, il doit conserver des mécanismes qui ne sont, dès lors, plus optimaux,...

L'objectif de notre sujet d'étude est de trouver une technique permettant la traduction du code machine x86, idéalement issu de la quinzième génération de processeurs x86, en clauses de Horn et ceci, sans aucune garantie de faisabilité. Nous nous focaliserons donc principalement sur le code machine x86, plutôt que sur la conception des processeurs x86 et les optimisations réalisées par ceux-ci.

1.2 Les clauses de Horn

1.2.1 Définition

Les clauses de Horn ont vu le jour en 1951 grâce à Monsieur Alfred Horn qui mit en évidence tout leur intérêt dans le monde de la logique mathématique dans l'article *On sentences which are true of direct unions of algebras* publié dans la 16ème édition du *Journal of Symbolic Logic* [12].

Avant de définir la notion de clause de Horn, revenons brièvement sur la notion de clause. Une clause est une disjonction de littéraux telle que

$$L_1 \vee ... \vee L_n$$

où L_i est un atome ou la négation d'un atome [14].

Les clauses de Horn, comme leur nom le laisse deviner, sont des clauses qui respectent une forme spécifique. En effet, il s'agit d'une clause comportant au maximum un littéral positif [12]. Il existe donc trois formes de clauses de Horn :

1. clause de Horn stricte : elle comporte un littéral positif et au moins un littéral négatif
2. clause de Horn positive : elle comporte un littéral positif et aucun littéral négatif
3. clause de Horn négative : elle ne comporte que des littéraux négatifs

L'une des caractéristiques principales de ces clauses de Horn est que leur forme normale disjonctive peut s'écrire sous forme d'une implication logique, comme le montre l'exemple suivant.

Clause de Horn stricte :

$$(\neg p \vee \neg q \vee r) \iff ((p \wedge q) \Rightarrow r)$$

Clause de Horn positive :

$$(p \vee q \vee r) \iff ((\neg p \wedge \neg q) \Rightarrow r)$$

Clause de Horn négative :

$$(\neg p \vee \neg q \vee \neg r) \iff ((p \wedge q) \Rightarrow \neg r)$$

Intuitivement, la clause de Horn stricte représente une implication permettant de déduire de nouveaux faits, la clause de Horn positives représente de simples faits et la clause de Horn négative représente une question et plus précisément, un but recherché.

1.2.2 Programmation logique par contraintes

Nous pouvons remarquer que les formes de clauses précédemment citées sont celles utilisées par le langage de programmation logique, Prolog. En effet, le principe de fonctionnement de Prolog est de résoudre une clause de Horn négative (recherche d'un but) avec une clause de Horn stricte afin de produire une autre clause de Horn négative, ce principe de résolution est appelé *SLD-resolution* [10]. Prolog est donc un langage de programmation déclarative qui repose sur la logique des prédicats restreinte aux clauses de Horn [13].

Exemple illustratif Des clauses de Horn aux clauses Prolog

Forme logique	$\forall x(\exists z, P(z,x) \wedge Q(z)) \Rightarrow Q(x))$
Forme clausale (Horn)	$Q(x) \vee \neg P(z,x) \vee \neg Q(x)$
Clause Prolog	*q(X) :- p(Z,X), q(Z).*

Cependant, nous avons décidé d'utiliser un cas particulier des clauses de Horn afin de faciliter le processus de traduction et d'optimiser l'exécution d'algorithmes Prolog. Il s'agit de la Programmation Logique par Contraintes (PLC), ou *Constraint Logic Programming (CLP)*.

La programmation logique par contraintes est une combinaison de deux paradigmes déclaratifs : la résolution de contrainte et la programmation logique [16]. Cette combinaison permet de rendre la résolution de problème syntaxiquement plus simple et, dans certains cas, d'obtenir une efficacité de résolution plus grande que dans d'autres paradigmes.

Nous n'expliquerons pas en détails la programmation logique par contraintes car ce mémoire se base principalement sur la traduction du code x86 en clauses PLC et non pas sur l'analyse de celles-ci. Les documents [16], [15] et [26] fournissent toutes les informations nécessaires à ce sujet et le document [26] est particulièrement pertinent car le sujet principal est directement lié à celui-ci.

Notre sujet d'étude étant intrinsèquement lié à ceux de Gonzague Yernaux [26] et Gérôme Laffineur [17] et la programmation logique par contraintes étant un paradigme de programmation, nous avons défini une syntaxe conceptuelle pour écrire les clauses de nos programmes PLC, ceci afin de les rendre plus facilement lisibles et compréhensibles. Notre syntaxe conceptuelle se rapproche fortement de celle de Prolog car cela permet de lui donner un aspect concret.

Ainsi, dans notre travail, une clause sera toujours constituée d'**une tête** et d'un **corps** composé d'**une contrainte facultative** et de prédicats. Dans la suite de ce document, nous appellerons nos programmes composés de telles clauses, des **programmes CLP**.

Exemple Clauses CLP.

```
p(A,B) :- {A > 3}, then(A-3,B).
p(A,B) :- {EAX =< 3}, else(A+3,B).
```

La syntaxe que nous utilisons se veut simple et facilement compréhensible. Ainsi, les contraintes seront toujours représentées en début de corps de clause et seront entourées par des accolades. L'exemple ci-dessus illustre un *IF-THEN-ELSE* avec comme condition $A > 3$.

Enfin, nous avons donc choisi d'utiliser les clauses Horn dans notre recherche car, étant à l'origine de Prolog, elles permettent d'effectuer des preuves automatiques de théorèmes avec une relativement bonne efficacité, ce qui se prête bien aux analyses de programmes. L'étude scientifique autour des clauses de Horn est présente et plusieurs méthodes de résolution ont donc déjà été étudiées. A titre d'exemple, le problème de satisfaisabilité appliqué aux clauses de Horn, appelé *HORN-SAT*, fait partie de la classe *P-complet* et est complet pour cette classe[7]. Les clauses CLP que nous utiliserons sont donc un cas particulier (dérivé) des clauses de Horn.

1.3 Objectif fixé

L'objectif de ce mémoire est de parvenir à mettre en place une technique qui permettrait de transformer un programme exécutable x86 en son équivalent en clauses de Horn (et plus précisément en CLP). L'approche proposée dans ce mémoire est de récupérer le code assembleur du fichier binaire et ensuite, d'effectuer une traduction séquentielle des ces instructions x86 en clauses CLP. En effet, le code assembleur x86 n'est autre qu'un code machine, autrement dit, il s'agit d'une suite d'instructions, manipulant des registres et des accès mémoire, exécutées séquentiellement par le processeur.

Durant ce projet, nous nous sommes également fixé l'objectif d'implémenter un outil permettant d'automatiser l'approche proposée dans ce mémoire afin de transformer un fichier exécutable en un ensemble de clauses de Horn. Cet outil, nommé *Hornix*, a été développé en Java et consiste donc à prendre en entrée un fichier binaire au format ELF et de produire en sortie son équivalent en clauses de Horn. Nous verrons par la suite l'architecture et les concepts utilisé dans l'implémentation de cet outil, ainsi que ses limitations.

1.4 Intérêts et contributions

Le principal objectif de ce sujet de recherche est de pouvoir appliquer aux programmes exécutables x86 une bonne partie des études et analyses de programmes réalisées sur les clauses de Horn. A titre de d'exemple, nous pourrions donc appliquer une analyse de terminaison de programme [3] sur des clauses de Horn issues d'un programme exécutable x86, ou encore une analyse de satisfaisabilité [7].

Outre le fait de bénéficier des études des clauses de Horn, nous pouvons également voir ces dernières comme un langage intermédiaire [11] dans le cas d'une traduction du langage initial en un autre langage de programmation.

En effet, la syntaxe et la sémantique des clauses de Horn permettent de les traduire sans trop de difficulté en langage de bas niveau comme de haut niveau. Ceci accorde un aspect universel et pourrait permettre de résoudre plusieurs problèmes actuels comme, par exemple, le support de la rétrocompatibilité par le code machine x86 précédemment énoncé.

Enfin, notre recherche peut également avoir un intérêt non négligeable dans le domaine de la rétro-ingénierie car les personnes travaillant et cherchant dans ce domaine sont souvent confrontées à devoir analyser du code binaire/hexadécimal ou du code assembleur. Nous pouvons imaginer divers outils permettant de faciliter leurs analyses en se basant sur les clauses de Horn issues des exécutables x86 étudiés. La rétro-ingénierie est également un sujet très étudié dans la sécurité informatique.

1.5 État de l'art et travaux relatifs

Le domaine d'étude dans lequel nous travaillons est malheureusement très peu étudié dans le domaine des Sciences Informatiques. En effet, concernant l'aspect clauses de Horn de notre travail, la communauté scientifique experte dans le domaine de la programmation logique reste très réduite et concernant le code machine x86, ce dernier est tellement vaste et complexe que très peu de scientifiques désirent investir de leur temps dans son étude.

Ce mémoire n'est d'ailleurs, ni plus ni moins, qu'une preuve de concept. En d'autres termes, il est actuellement impossible de savoir si une transformation totale d'un programme exécutable x86 en clauses de Horn est réalisable ou non. Tout l'intérêt de ce mémoire est justement de savoir si cela est possible, moyennant certaines contraintes, ou non.

Cependant, certains travaux scientifiques plus ou moins en relation avec le notre et peuvent nous servir d'appui et d'autres nous fournissent des informations cruciales concernant certains aspects de nos recherches.

Premièrement, [19] est un document réalisé dans le même projet de recherche que celui dans lequel nous travaillons. Les auteurs décrivent une méthode permettant de traduire un ensemble restreint de programmes écrits en bytecode java et destinés à être interprétés par la machine virtuelle Dalvik [2]. Cette recherche est donc fort similaire à la notre et prouve encore une fois l'efficacité et l'intérêt des clauses de Horn dans cet objectif de traduction de programme.

Deuxièmement, [11] est une publication démontrant l'universalité des clauses de Horn comme langage intermédiaire dans les analyses et transformations de programmes. Cette publication abordent donc l'intérêt et l'efficacité des clauses de Horn dans le processus de compilation d'un programme et a donc été cruciale dans notre choix des clauses de Horn au sein de notre recherche.

Enfin, d'autres recherches comme [22] se sont attardées sur la décompilation de programmes exécutables x86. Bien que leur objectif était axé sur le fait de

retrouver le code source original, ce qui n'est pas notre cas, leurs conclusions ne nous en sont pas moins utiles et nous démontrent la difficulté de réaliser certaines opérations ou analyses sur le code machine x86, voir leur impossibilité. Certaines recherches comme [21] démontrent même que le simple fait de désassembler un fichier binaire en code assembleur présente des erreurs et perd, presque de façon systématique, des informations.

Chapitre 2

Background technique sur le code machine x86

Dans cette section, nous expliquons certains concepts de base qui sont indispensables à la bonne lecture de ce mémoire. Nous n'expliquerons donc pas tous les concepts du code machine x86 et certains d'entre eux seront également expliqués plus tardivement dans le mémoire afin de rendre la lecture plus aisée.

Enfin, le code machine x86 ayant subi une évolution permanente durant plus de 20 ans, nous nous baserons dans ce mémoire sur le code machine x86 actuel, généralement appelé le *x86 moderne*, dans sa version **32 bits**.

2.1 Les conventions d'écriture

Il existe deux conventions d'écriture concernant le code machine x86 : Intel et AT&T. Il est important de noter que celles-ci n'engendrent aucun changement dans l'exécution du code, il s'agit de simples conventions d'écriture qui sont chacune utilisée par divers outils.

Préfixe Dans la convention d'écriture AT&T, les opérandes des instructions sont préfixés par "%" s'il s'agit de registres ou de "$" s'il s'agit de valeurs immédiates [1]. *A contrario*, la syntaxe Intel ne nécessite pas de préfixe excepté lorsqu'un nombre hexadécimale commence par un lettre, dans ce cas, il convient de préfixer ce nombre par 0.

Suffixe Une des majeures différences entre ces deux syntaxes est que AT&T suffixe les opérateurs en fonction de la taille des opérandes : *l* pour *long*, *w* pour *word* et *b* pour *byte*.

La syntaxe Intel possède également des suffixe (*byteptr*, *wordptr*, *dwordptr*) mais ceux-ci sont facultatifs puisque les registres utilisés sont supposés être de la taille adéquate.

1. Une valeur immédiate est une valeur directe

Registe de travail	Rôle
EAX	**A**ccumulateur, il est utilisé pour réaliser opérations arithmétiques
EBX	**B**ase, il est utilisé pour l'adressage indirect
ECX	**C**ompteur, il est utilisé pour le comptage, lors d'un passage dans une boucle, par exemple
EDX	**D**onnée, il est utilisé en complément à EAX
Registe d'offset	Rôle
ESI	**S**ource **I**ndex, il est utilisé lors d'opérations sur des chaînes de caractères
EDI	**D**estination **I**ndex, il est utilisé en complément à ESI
ESP	**S**tack **P**ointer, il pointe vers le dernier élément empilé sur la pile
EBP	**B**ase **P**ointer, il référence la base du *stackframe* de la pile

TABLE 2.1 – Rôles des différents registres

Syntaxe des opérandes Au niveau syntaxique, les deux syntaxes sont relativement opposée. En effet, Intel positionne l'opérande de destination en première position tandis que AT&T positionne son opérande destination en dernière position.

Opérandes mémoire Les opérandes référençant à la mémoire sont entourés par des crochets sous la syntaxe Intel tandis que la syntaxe AT&T requiert des parenthèses.

Une autre différence réside dans les instructions mémoire complexes (*segreg*) mais nous n'en parleront pas dans ce mémoire.

2.2 Les registres

Les processeurs x86 travaillent avec huit registres de travail généraux et, historiquement, ceux-ci avaient chacun un but d'utilisation [6], comme le montre le tableau 2.1. Nous reviendrons sur les concepts de pile et de *stackframe* par la suite.

Ces buts d'utilisation ne sont plus d'actualité au jour d'aujourd'hui. Bien que ces registres soient optimisés pour réaliser leur tâche respective, rien n'empêche de les utiliser pour tout autre chose. Par exemple, le registre *ECX* est optimisé pour compter mais nous pouvons très bien l'utiliser pour réaliser une opération arithmétique. Seuls les registres *ESP* et *EBP* ont gardé leur convention d'usage.

Comme expliqué précédemment, les premiers processeurs x86 ne travaillaient qu'avec des registres d'une taille de 16 bits et, à cause de la rétro-compatibilité, ces registres de 16 bits sont toujours conservés dans le code machine x86 moderne, qui fonctionne habituellement avec des registres de 32 bits et 64 bits.
De ce fait, et comme illustré à la figure 2.1 [20], les registres de travail *E*X* de

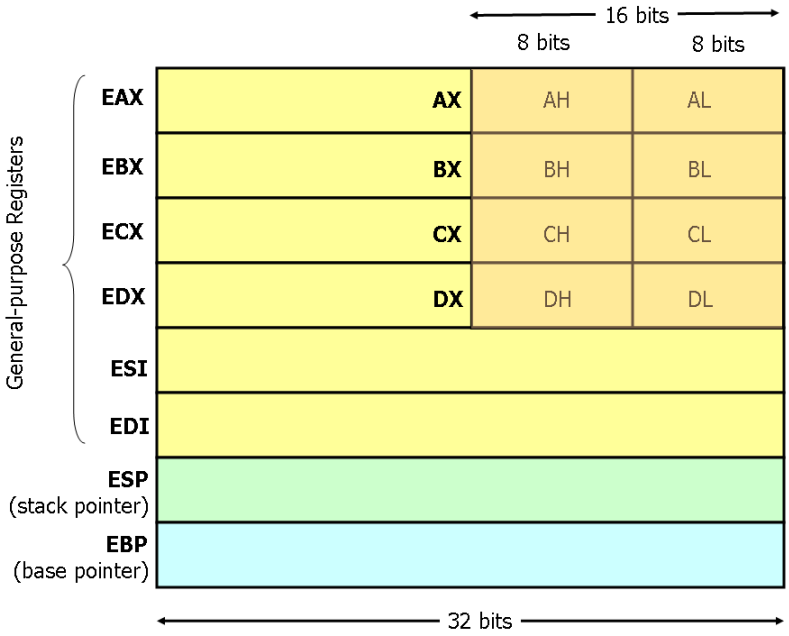


FIGURE 2.1 – Structure des différents registres

32 bits possèdent trois sections qui peuvent être utilisées par le code machine x86 et qui sont :

— *X : les 16 bits de poids faible du registre
— *L : les 8 bits de poids faible de la section de 16 bits
— *H : les 8 bits de poids fort de la section de 16 bits

Enfin, il existe également deux autres registres qui sont couramment utilisés mais qui sont également particuliers. Le premier est *EIP*, qui s'étend sur 32 bits et qui pointe sur la prochaine instruction à exécuter, il ne peut donc pas être modifié directement. Le second est *eflags* et est également composé de 32 bits. Sa particularité est que l'on y accède pas dans son intégralité mais bit par bit car chacun de ses bits, nommés *flags* ou *drapeaux*, contient une information spécifique, comme indiqué à la figure 2.2 [20].

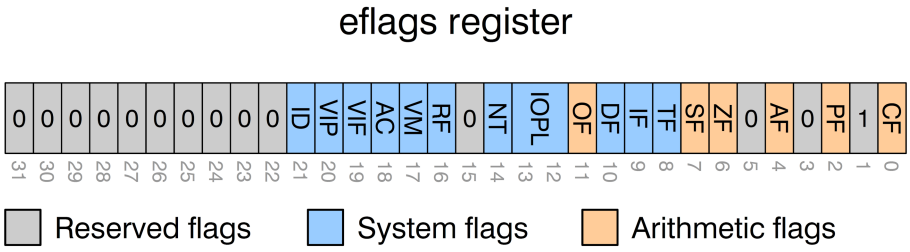


FIGURE 2.2 – Structure du registre *eflags*

Quelques drapeaux importants [6] :

Drapeau	Nom
CF	Carry Flag
PF	Parity Flag
AF	Auxiliary carry Flag
ZF	Zero Flag
SF	Sign Flag
IF	Interruption Flag
DF	Direction Flag
OF	Overflow Flag

2.3 La mémoire

2.3.1 La pile et l'adressage direct

Il va de soi qu'un processeur muni de huit registres ne permet pas de réaliser un nombre conséquent d'opérations et n'est pas très performant. C'est pourquoi un module physique, appelé mémoire RAM (*Random Acces Memory*) vient s'ajouter à la puissance de calcul du processeur afin de lui fournir un plus grand espace mémoire.

L'accès à la mémoire RAM est moins rapide que l'accès aux registres mais reste toute fois très rapide (de l'ordre de la nanoseconde) et c'est pourquoi elle est utilisée en permanence par le processeur. Le code machine x86 prévoit deux façons d'accéder à la mémoire RAM.

La première façon est une représentation d'une partie de la mémoire par une pile fonctionnant sous le principe de la méthode LIFO (*Last In, First Out*), cette pile peut être assimilée à un tableau d'emplacements de 32 bits. Tout logiquement, l'ensemble d'instructions x86 prévoit deux instructions afin de pouvoir écrire et lire sur la pile, il s'agit des instructions *PUSH* et *POP*. Avant d'expliquer ces deux instructions, rappelons-nous que le registre *ESP* pointe vers le dernier élément empilé sur la pile.

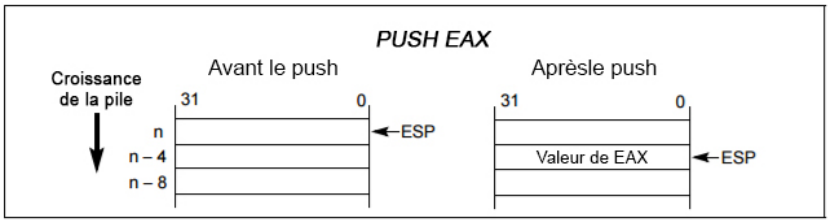


FIGURE 2.3 – Représentation de l'instruction *push eax*

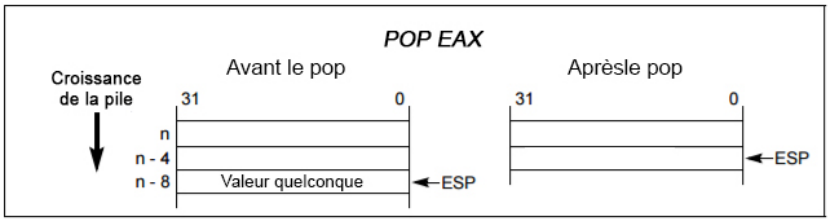


FIGURE 2.4 – Représentation de l'instruction *pop eax*

L'instruction *PUSH* décrémente, dans un premier temps, le registre *ESP* de 4 octets (32 bits) et empile ensuite la valeur de son opérande, comme illustré par la figure 2.3.

L'instruction POP, quant à elle, effectue l'opération inverse, c'est-à-dire qu'elle copie la valeur pointée par *ESP* dans l'opérande de destination et ensuite, incrémente le registre *ESP* de 4 octets, comme illustré par la figure 2.4.

La deuxième façon d'accéder à la mémoire est un accès directe à celle-ci via une adresse mémoire. En effet, en entourant l'adresse mémoire désirée par des crochets, nous informons le processeur que nous voulons accéder directement à un endroit particulier de la mémoire.

Exemple

- *mov eax, [0DEADBEEF]* : copie la valeur se trouvant dans la mémoire à l'adresse 0xDEADBEEF dans *EAX*
- *mov [eax], ebx* : copie la valeur de *EBX* dans la mémoire à l'adresse contenue dans *EAX*
- *mov eax, [esp+4]* : copie la valeur se trouvant dans sur la pile à l'adresse *ESP+4*, c'est-à-dire l'avant dernière valeur de la pile, dans *EAX*

Enfin, notons que les bonnes pratiques veulent que lorsqu'une valeur est temporairement empilée sur la pile, celle-ci soit désempilée par la suite. Afin d'optimiser les ressources et l'efficacité, l'instruction *POP* étant peu performante, on préfère directement incrémenter le registre *ESP* afin de restituer la pile à l'état initial. Ainsi, nous effectuerons l'opération *add esp, 8* afin de "désempiler" les deux derniers éléments empilés sur la pile. On utilise 8 car on supprime deux fois 4 octets.

2.3.2 Les instructions

Cette section n'a pas pour but de répertorier toutes les instructions du code machine x86, celles-ci étant listées dans le document *Architectures Software Developer Manuals* [6] d'Intel, mais plutôt d'informer sur les différents types d'instructions qui seront majoritairement traités dans ce mémoire.

Instructions arithmétiques L'ensemble d'instructions x86 possède, tout logiquement, des instructions permettant de réaliser des opérations arithmétiques telles que l'addition, la soustraction, l'incrémentation, la décrémentation,... Celles-ci sont majoritairement utilisées dans les programmes.

Les opérations logiques sont également disponibles (*AND, OR, NOT,...*) ainsi que des opérations sur les bits des registres, comme *shift-left* et *shift-right* (*shl, shr*).

Instructions de déplacement de donnée Ces instructions permettent principalement d'affecter des données à des registres (*mov*), d'empiler et désempiler des données sur la pile (*push, pop*) et de manipuler des adresses mémoires (*lea*).

Instructions de branchement Ce type d'instruction intervient pour palier au fait que l'exécution du code assembleur est séquentielle. En effet, certaines instructions permettent d'effectuer ce que l'on appelle des sauts conditionnels et inconditionnels (*jmp, je, jne, jg,...*) et d'appeler des fonctions (*call, ret*).

2.4 Les fonctions

L'usage de fonctions est un concept primordial en programmation et ce concept est également présent dans le code machine x86. Le code machine, étant un langage de bas niveau, est beaucoup plus permissif que les langages de programmation de haut niveau. Cela signifique que tout programmeur de code x86 est libre de créer ses fonctions comme bon lui semble et c'est pourquoi il existe plusieurs conventions d'appel de fonction. [8] liste une multitude de conventions utilisées par la plupart des compilateurs connus et, dans le cadre de notre projet, nous nous baserons que sur une seule de celles-ci. La convention d'appel de fonction que nous respecterons est celle qu'utilise la série d'outils *GNU* et est maintenant presque utilisée par quasiment tous les programmeurs et respectée par de nombreux outils touchant à l'assemblage de programme.

L'existence de convention d'appel de fonction est indispensable car l'utilisation de variables locales,... risque de laisser la pile dans un état inconsistant après l'exécution de la fonction appelée. Cette section a donc pour but d'expliquer cette convention d'appel de fonction en x86.

L'idée principale de cette convention d'appel est d'allouer un espace d'adressage (appelé *stackframe*[2]) dans la pile pour la fonction appelée et de supprimer cet espace d'adressage à la fin de celle-ci. De ce fait, la pile sera revenue à son état initial et le programme principal (ou la fonction appelante) pourra continuer son exécution.

2. Une stackframe un donc relatif à une fonction ou au programme principal. Le stackframe courant est celui délimité par *EBP* et *ESP*

Cette convention d'appel de fonction peut être représentée par deux ensembles de règles à respecter. Le premier étant pour l'appelant et le second pour l'appelé. Avant d'expliciter ces règles, rappelons que le registre *EBP* pointe vers la base du stackframe courant et que le registre *ESP* pointe vers le dernier élément empilé sur celui-ci.

Les règles que nous allons maintenant décrire se base sur l'hypothèse que nous travaillons avec des arguments de 32 bits.

Nous listons ci-dessous les règles se rapportant à l'appelant, c'est-à-dire au programme (ou fonction) appelant une autre fonction.

1. La première règle est facultative. En effet, si l'appelant contient des données dans des registres devant être utilisées après l'appel de l'appelé, il est nécessaire de les empiler sur la pile afin des les restituer après l'exécution de la fonction appellée. Ainsi cette dernière pourra également utiliser les registres et nous pourrons retrouver leur valeur initiale après son exécution.
2. Afin de fournir les paramètres nécessaires à la fonction appelée, l'appelant doit les empiler sur la piles et ce, dans l'ordre inverse (c'est-à-dire le dernier argument en premier lieu).
3. Une fois le passage des arguments effectués, l'appel de la fonction se fait par l'instruction *CALL*. Cette instruction va implicitement empiler la valeur du registre *EIP* et dans ce cas, il s'agit de l'adresse de l'instruction suivant l'appel de fonction.
4. Lorsque l'appelé a terminé son exécution tout en respectant les règles qui s'appliquent à son cas, l'appelant doit supprimer les arguments précédemment empilés sur la pile afin de retrouver l'état initial de celle-ci. Ceci se fait via l'instruction *ADD ESP, 4*X*[3] où X représente le nombre de paramètre.
5. Enfin, si certains registres avaient été "sauvegardés", il faut donc les restaurer à l'aide de l'instruction *POP*.

Nous listons maintenant ci-dessous les règles se rapportant à l'appelé, c'est-à-dire à la fonction qui a été appelée par l'appelant.

1. Lorsque l'exécution de la fonction appelée peut commencer, celle-ci doit, avant tout, créer son espace d'adressage sur la pile. Pour cela, elle devra empiler la base de la pile (*push ebp*) sur la pile et ensuite, assigner la valeur de *ESP* à *EBP* (*mov ebp, esp*).
   Cela permet de sauvegarder la valeur de *EBP* en vue de la restaurer par la suite et de définir une nouvelle base de la pile au sommet de celle-ci. Par convention, la fonction ira chercher les paramètres passés par l'appelant par rapport à EBP, ainsi le premier paramètre se trouvera à l'adresse *EBP + 4*.
2. Une fois fait, la fonction appelée doit maintenant définir un espace pour ses variables locales. Pour ce faire, il suffit de décrémenter *ESP* de 4*X où X est le nombre de variables locales, comme déjà vu.

3. Car nous travaillons avec paramètres soient de 32 bits

3. À ce stade-ci, le corps de l'appelé peut s'exécuter. Après son exécution, l'appelé doit impérativement renvoyer son résultat final dans le registre *EAX*.
4. Une fois le résultat final placé dans *EAX*, il faut maintenant supprimer l'espace d'adressage créé pour les variables locales. Pour ce faire, nous pouvons incrémenter *ESP* comme nous l'avons vu ou, plus simplement, assigner la valeur de *EBP* à *ESP* (*mov esp, ebp*).
5. Enfin, l'appelé doit maintenant restituer l'ancienne valeur de *EBP* en utilisant l'instruction *pop ebp* et doit renvoyer le flux d'exécution à l'appelant via l'instruction *RET*. Implicitement, cette instruction désempile la valeur de *EIP* précédemment empilée sur la pile par l'instruction *CALL* et l'assigne à *EIP*.

Exemple Soient P un programme et *maFonct* une fonction appelée par P qui calcule la somme des deux paramètres reçus en entrée.

Code C du programme$P$:

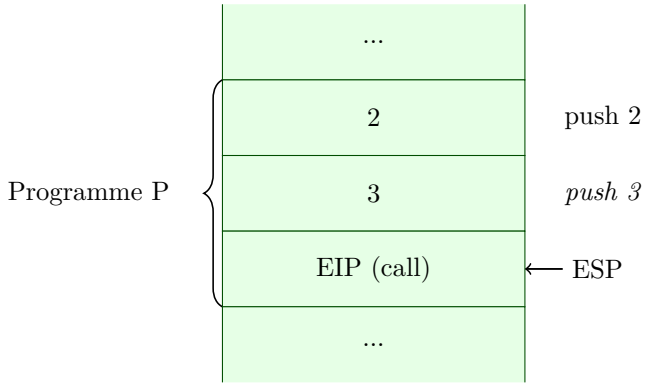
```
int main() {
    int a = 3;
    int b = 2;
    return maFonct(a,b);
}

int maFonct(int x, int y) {
    return x + y;
}
```

Code assembleur x86 du programme P :

```
...
push 0x2 ; Push du deuxième paramètre (2), écrit en hexadécimal
push 0x3 ; Push du premier paramètre (3), écrit en hexadécimal
call maFonct ; Appel de la maFonct
add esp, 0x8 ; Suppression des paramètres précédemment empilés
...
```

État de la pile après l'instruction *CALL* :



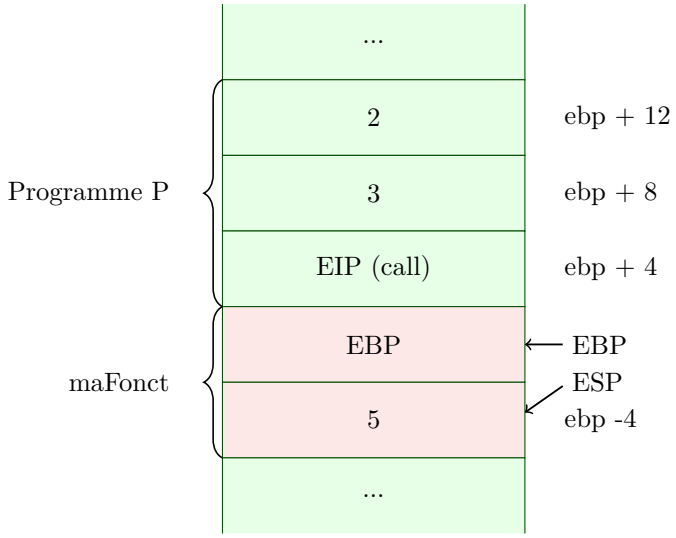
Code machine x86 de la fonction *maFonct* :

```
maFonct: ; Label qui détermine la fonction "maFonct"
 push ebp ; sauvegarde de EBP
 mov ebp, esp ; définition de la base du stackframe de maFonct

 sub esp, 4 ; réservation de l'espace pour la variable locale
 mov eax, [ebp+12] ; eax = 3
 mov edx, [ebp+8] ; edx = 2
 mov [ebp-4], eax ; variable locale = eax
 add [ebp-4], edx ; variable locale = variable locale + edx
 mov eax, [ebp-4] ; résultat final placé dans eax

 mov esp, ebp ; suppression du stackframe de maFonction
 pop ebp ; restauration de l'ancien EBP
 ret ; retour à l'appelant
```

État de la pile avant la suppression du stackframe de *maFonct* :



2.5 L'évolution x86-64

Depuis quelques années, et plus précisément, depuis le début des années 2000, les processeurs ont encore une fois évolué et sont passés d'un système 32 bits à 64 bits. Notre étude se basant sur des exécutables x86 en 32 bits (notés *x86-32*), nous allons brièvement décrire les impacts majeurs de cette évolution sur le code assembleur x86 (noté *x86-64*).

Tout comme l'évolution des processeurs 16 bits en 32 bits, les registres généraux ont été étendus à 64 bits. Les nouveaux registres sont nommés *RAX, RBX, RCX, RDX, RSP, RBP, RSI et RDI* et les anciens registres occupent donc les 32 bits de poids faible de ceux-ci. De plus, une autre différence majeure est que les nombres de registres généraux à lui-même été augmenté. En effet, huit nouveaux registres 64 bits ont été ajoutés : *R9, R10, R11, R12, R13, R14, R15*. Ceci permet donc d'augmenter la capacité de calcul du processeur et de réduire l'accès à la mémoire RAM. Notons que ces nouveaux registres peuvent également des sous-registres de 32, 16 et 8 bits comme illustré sur la figure 2.5 [20].

Enfin, il existe évidemment bien d'autres changements suite à cette évolution mais comme dit en début de section, nous ne nous en occuperons pas pour le moment. A titre d'exemple, les instructions arithmétiques peuvent s'opérer en 64 bits avec des opérandes de 32 bits car elles mettent automatiquement les 32 bits de poids fort des opérandes 64 bits à 0. Cependant, ce n'est pas le cas de toutes les instructions et par conséquent, les 32 bits de poids forts restent inchangés et ceci peut donc poser plusieurs problèmes dans la suite du programme.

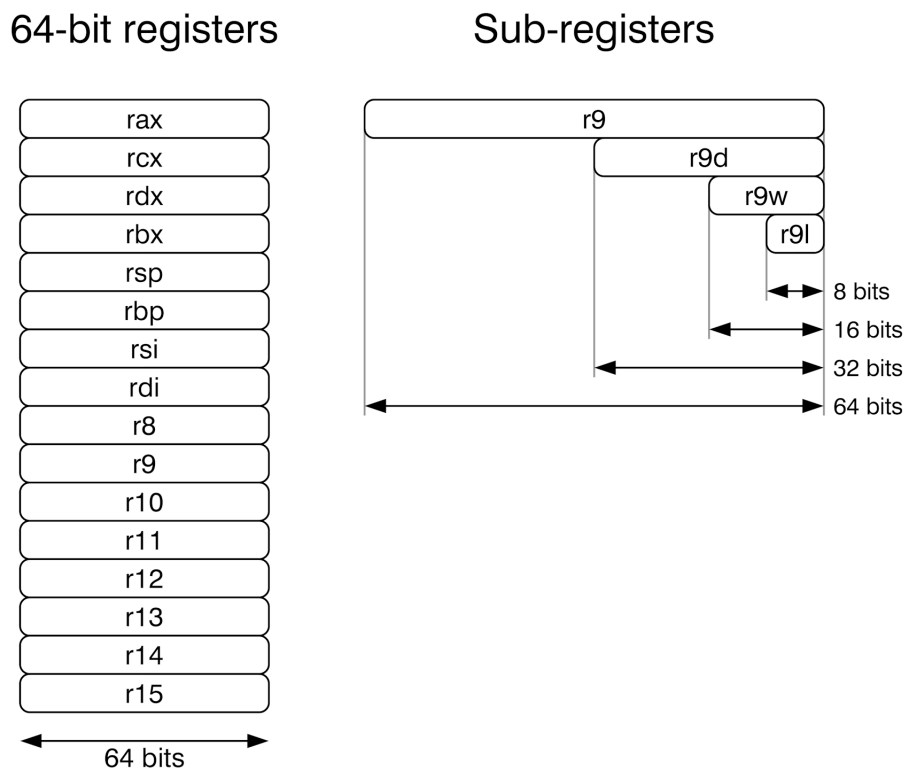


FIGURE 2.5 – Structure des registres 64 bits

Chapitre 3

Définition du modèle de traduction

3.1 Le format objet ELF

3.1.1 Définition

Avant de commencer nos recherches sur la traduction de fichiers exécutables x86 en clauses de Horn, il convient de fixer notre environnement de travail et plus précisément sur quel type de fichier que nous allons traiter.

En effet, les processeurs x86 sont présents dans la plupart des machines informatiques mais ces dernières peuvent tourner sous une large palette de systèmes d'exploitation tels que Windows, les systèmes UNIX, Mac OS X, ... Ceci implique que les fichiers binaires exécutables x86 se trouvent sous plusieurs formats différents en fonction de ces systèmes d'exploitation, nous pouvons citer le format ELF (*Executable and Linkable Format*) pour les systèmes UNIX, le format PE (*Portable Executable*) dont l'extension la plus connue est .exe, pour Windows et Mach-O pour Mac OS X.

Comme chaque format de fichier exécutable possède une structure différente, il n'est donc pas envisageable de tous les traiter dans ce mémoire et c'est pouquoi nous avons choisi de nous intéresser uniquement au format ELF pour les raisons suivantes :

1. il s'agit d'un des formats les plus répendus et qui est ouvert
2. il a l'avantage d'être mieux structuré

Il convient donc dans un premier temps de porter nos recherches sur la structure et la composition d'un fichier ELF afin de pouvoir en tirer les informations qui nous intéressent, c'est-à-dire principalement le code machine qui correspond au programme initial codé en langage haut niveau.

Le format ELF est donc un format de fichier exécutable binaire principalement utilisé sur les machines de type Unix, à l'exception de Mac OS. Grâce à sa flexibilité, son extensibilité et le fait qu'il soit un format ouvert, le format ELF ne dépend d'aucun processeur ou architecture [4].

Le format ELF permet d'encapsuler toutes les informations nécessaires à la bonne exécution du programme qu'il représente, comme par exemple l'espace mémoire à réserver pour la bonne exécution de ce dernier. Cette encapsulation permet au fichier ELF de représenter plusieurs types de fichier objet, tels que :

1. *Fichier objet repositionnable* : contient du code et des données destinés à être mis en relation avec d'autres fichiers de ce type afin de créer un fichier exécutable ou un fichier objet partagé.
2. *Ficher objet exécutable* : contient un programme destiné à être directement exécuté.
3. *Fichier objet partagé* : contient du code et des données destinés à être liés avec d'autres fichiers objets afin de créer un autre fichier objet ou à permettre l'exécution d'un fichier objet exécutable. Ce type représente souvent des librairies dynamiques devant être liées à l'exécutable lors de son exécution.
4. *Fichier objet core* : Plus rarement utilisé, ce type de fichier représente un état du fichier binaire à un moment défini.

Ces fichiers objets sont créés par l'assembleur et l'éditeur de lien et sont des représentations binaires de programmes ayant pour but d'être directement exécutés sur un processeur [4].

Dans le cadre de notre projet de recherche, notre attention sera principalement portée sur les fichers objets ELF exécutables.

3.1.2 Création (compilation)

Lorsque nous compilons un programme que nous avons écrit en langage haut niveau (en utilisant GCC [9] sur un programme écrit en C, par exemple), cela renvoie un output exécutable représentant notre programme initial. Cependant, afin de comprendre comment le fichier ELF est créé et à quel moment le langage assembleur intervient-il, il convient de comprendre ce processus de compilation. Ce processus se déroule en quatre étapes consécutives qui sont les suivantes.

Le pré-processeur La phase de pré-processing effectue un traitement sur le fichier contenant le programme écrit en langage haut niveau qu'il prend en input. Ce traitement consiste en toutes sortes de modifications telles que la suppression des commentaires, le remplacement des caractères spéciaux afin d'éviter tout problème dans les chaines de caractères, la vérification de l'absence d'erreurs de syntaxe,... Un des traitements importants durant cette phase est la résolution et l'inclusion des headers (prototypages) utilisés dans le programme (appels à d'autres fichiers sources). L'outil GNU pour réaliser cette étape est *CPP* [9].

La compilation Une fois que nos fichiers sources ont été pré-traités, on ne traduit pas directement le code haut niveau en binaire mais on le traduit en un code de bas niveau, appelé code assembleur. L'étape de compilation vérifie donc l'absence d'erreur sémantique et autre via des arbres syntaxiques, tables des symboles,... [1] pour finalement traduire le code source en code assembleur afin de pouvoir simplifier la traduction vers le code binaire. L'outil GNU pour réaliser cette étape est *GCC* [9].

L'assemblage La traduction du code assembleur vers le code binaire est appelée l'assemblage. Après cette étape, nous obtenons donc un fichier binaire à partir d'un fichier assembleur, il serait donc intéressant, dans le cadre de notre projet de pouvoir effectuer l'étape inverse afin d'obtenir un code assembleur à partir d'un fichier binaire. Nous verrons par la suite que ce processus existe et est appelé désassemblage. L'outil GNU pour réaliser l'étape d'assemblage est *AS* [9].

La liaison Après l'étape d'assemblage, notre fichier binaire n'est toujours pas fini. En effet, les librairies externes ont été prototypées mais leur implémentation ne se trouve pas dans le fichier binaire. Une étape, dite de liaison, est donc nécessaire afin de lier à notre fichier binaire, toutes les fonctions/librairies externes auxquelles il fait appel. On distingue deux types de liaison :

- liaison statique : le code des fonctions/librairies externes est incorporé dans le fichier binaire
- liaison dynamique : Des informations permettant de savoir quelles fonctions/librairies externes sont nécessaires à la bonne exécution du programme sont incorporées dans le fichier binaire à la place du code de celles-ci. C'est donc à l'exécution du programme que le processeur ira chercher dans la mémoire de la machine le code des fonctions adéquates.

La liaison statique était anciennement utilisée et a laissé place à la liaison dynamique qui permet d'obtenir des fichiers binaires moins lourds car ils ne contiennent plus le code des fonctions/libraires externes. L'outil GNU pour réaliser cette étape est *LD* [9].

3.1.3 Structure

Introduction Un fichier ELF, une fois compilé, contient toute les informations nécessaires à sa bonne exécution cependant, ces informations sont disséminées à plusieurs endroits définis dans le fichier. Il est donc impossible d'effectuer une lecture linéaire d'un fichier ELF. De ce fait, tout fichier ELF possède une structure spécifique qui permet de retrouver les informations désirées, nous allons décrire cette architecture sans pour autant rentrer dans les détails.

Cette structure consistera toujours en un entête ELF (appelée *ELF Header*) suivie d'une table des entêtes de programme (*Program header table*) suivie de segments contenant une ou plusieurs sections et se terminant par une table des entêtes de sections (*Section header table*) [4], comme représenté sur la figure 3.1.

ELF Header L'ELF Header est présent au début (offset de 0) de chaque fichier ELF et contient des informations sur l'organisation du fichier indispensables à la lecture/exécution. Le tableau 3.1 [4] reprend ces informations.

Segment Comme énoncé précédemment, le système utilise un procédé de segmentation pour accéder à la mémoire. Sans rentrer dans les détails, lorsque des adresses mémoire sont utilisées dans des programmes C, par exemple, celles-ci sont virtuelles et le système se charge de retrouver l'adresse physique exacte en fonction de son système de segmentation de la mémoire. Pour plus d'information sur ce mécanisme, le manuel du développeur sur l'architecture Intel [5] détaille beaucoup plus précisément ce principe.

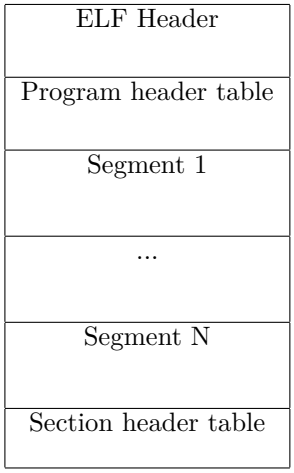


FIGURE 3.1 – Structure générale d'un fichier ELF

Nom du champ	Description
e_ident	Tableau permettant d'identifier le fichier binaire et de vérifier qu'il est bien un fichier ELF (32 ou 64 bits)
e_type	Défini le type du fichier objet
e_machine	Défini l'architecture de la machine nécessaire à l'exécution du fichier
e_versus	Défini la version actuelle du fichier
e_entry	Défini l'adresse virtuelle à laquelle démarre le processus (transfère du contrôle par le système)
e_phoff	Contient le décalage en octets de la table contenant l'entête de programme
e_shoff	Contient le décalage en octets de la table des entêtes de sections
e_flags	Contient des drapeaux spécifiques au processeur
e_ehsize	Contient la taille de l'entête ELF en octets
e_phentsize	Contient la taille en octets d'une entrée de la table d'entête de programme. Toutes les entrées ont la même taille
e_phnum	Contient le nombre d'entrées de la table d'entête de programme
e_shentsize	Contient la taille en octets d'un entête de section (un entête de section est une entrée de la table des entêtes de sections). Toutes les entrées ont la même taille
e_shnum	Contient le nombre d'entrées de la table des entêtes de sections
e_shstrndx	Contient l'indice dans la table des entêtes de sections de l'entrée associée à la table des chaînes de noms des sections

TABLE 3.1 – Structure du ELF Header

L'important à savoir sur ces segments est que le fichier binaire ELF est réparti sur plusieurs segments de la mémoire et chacun de ces segments possèdent ses propres droits et propriétés. Ainsi, nous retrouvons le système de droits des fichiers (*read*, *write*, *execute*) sur les segments et il peut être intéressant de noter que les segments possédant le droit d'exécution seront probablement les segments qui contiendront du code interprétable par le processeur.

Program header table La table d'entête de programme d'un fichier exécutable ou objet partagé est un tableau de structure, où chacune d'entre elles contient des informations sur un segment ou d'autres informations sur nécessaire à l'exécution du programme [4]. En effet, lorsqu'un programme est exécuté, le système charge ce programme dans la mémoire et créé, ce que l'on appelle une image processus. Le système travaillant avec un mécanisme de segmentation de la mémoire, le *program header* table lui permet donc de pouvoir représenter et manipuler correctement le programme dans la mémoire organisée en segment. Ce point sera un peu plus détaillé au paragraphe suivant.

Ce tableau de structure se présente de la même façon que l'*ELF Header* mais nous ne nous attarderons pas sur toutes les informations contenues dans ce tableau, référez-vous au manuel EFL [4] pour plus de précision. Une des informations importantes de l'entête de programme est *p_type* car cette dernière nous informe sur le type de chaque segment. En effet, il existe plusieurs type de segment dont deux qui nous intéressent :

- PT_LOAD : indique un segment chargeable (c'est souvent ici que nous retrouverons le code exécutable)
- PT_DYNAMIC : contient des informations de liaison dynamiques (librairies externes)

Sections Comme leur nom l'indique, les sections sont des sections du fichiers binaires, elles composent donc les segments. Comme déjà dit, les fichiers ELF possèdent des informations de différents types, comme par exemple du code exécutable, des informations sur les symboles, les librairies,... Ces informations sont justement organisées à travers ces sections. Ainsi, comme indiqué sur la figure 3.1, en fin de fichier ELF, nous retrouvons une table d'entêtes de sections (*section header table*) qui nous fournit de multiples informations sur ces sections, comme leur type, leur taille, leur position,... Parmi l'ensemble de ces sections, certaines d'entre elles vont particulièrement nous intéresser.

La section *.text* est sûrement la plus importante de toutes car c'est dans cette section que nous allons retrouver le code exécutable du fichier binaire. Ci-dessous un extrait du résultat obtenu lorsque nous désassemblons la section *.text* du programme C de la section 2.4. Nous nous attarderons plus précisément sur ce code par la suite, pour le moment il est intéressant de noter que nous pouvons distinguer la fonction *main* et la fonction *maFonct* dans ce code.

```
080482f0 <_start>:
 80482f0: 31 ed                 xor    ebp,ebp
 80482f2: 5e                    pop    esi
 80482f3: 89 e1                 mov    ecx,esp
 80482f5: 83 e4 f0              and    esp,0xfffffff0
```

```
 80482f8: 50                    push   eax
 80482f9: 54                    push   esp
 80482fa: 52                    push   edx
 80482fb: 68 a0 84 04 08        push   0x80484a0
 8048300: 68 30 84 04 08        push   0x8048430
 8048305: 51                    push   ecx
 8048306: 56                    push   esi
 8048307: 68 ed 83 04 08        push   0x80483ed
 804830c: e8 cf ff ff ff        call   80482e0 <__libc_start_main@plt>
 8048311: f4                    hlt
 8048312: 66 90                 xchg   ax,ax
 8048314: 66 90                 xchg   ax,ax
 8048316: 66 90                 xchg   ax,ax
 8048318: 66 90                 xchg   ax,ax
 804831a: 66 90                 xchg   ax,ax
 804831c: 66 90                 xchg   ax,ax
 804831e: 66 90                 xchg   ax,ax
...
080483ed <main>:
 80483ed: 55                    push   ebp
 80483ee: 89 e5                 mov    ebp,esp
 80483f0: 83 e4 f0              and    esp,0xfffffff0
 80483f3: 83 ec 20              sub    esp,0x20
 80483f6: c7 44 24 18 02 00 00  mov    DWORD PTR [esp+0x18],0x2
 80483fd: 00
 80483fe: c7 44 24 1c 03 00 00  mov    DWORD PTR [esp+0x1c],0x3
 8048405: 00
 8048406: 8b 44 24 1c           mov    eax,DWORD PTR [esp+0x1c]
 804840a: 89 44 24 04           mov    DWORD PTR [esp+0x4],eax
 804840e: 8b 44 24 18           mov    eax,DWORD PTR [esp+0x18]
 8048412: 89 04 24              mov    DWORD PTR [esp],eax
 8048415: e8 02 00 00 00        call   804841c <maFonct>
 804841a: c9                    leave
 804841b: c3                    ret

0804841c <maFonct>:
 804841c: 55                    push   ebp
 804841d: 89 e5                 mov    ebp,esp
 804841f: 8b 45 0c              mov    eax,DWORD PTR [ebp+0xc]
 8048422: 8b 55 08              mov    edx,DWORD PTR [ebp+0x8]
 8048425: 01 d0                 add    eax,edx
 8048427: 5d                    pop    ebp
 8048428: c3                    ret
 8048429: 66 90                 xchg   ax,ax
 804842b: 66 90                 xchg   ax,ax
 804842d: 66 90                 xchg   ax,ax
 804842f: 90                    nop

 ...
```

Ensuite, les sections *.data, .rodata et .bss* sont également importantes car elles détiennent respectivement les variables globales et statiques explicitement initialisées dans le code source, les constantes et les variables non-initialisées.

3.1.4 *OBJDUMP* comme outil de désassemblage

Maintenant que nous avons observé et mieux compris la structure et la composition d'un fichier ELF, nous savons que nous ne pouvons pas le lire séquentiellement en vue de le transformer en clauses de Horn. Afin de pouvoir l'interpréter et le traduire, l'idéal serait de pouvoir le décompiler, c'est-à-dire retrouver le code source original mais nous venons de voir que la compilation est un concept très complexe et malheureusement, la décompilation l'est encore plus et reste une sujet de recherche plein d'énigmes [22].

Nous avons donc décidé de nous baser sur le code assembleur x86 que nous pouvons facilement obtenir *via* plusieurs outils, nous allons donc effectuer un désassemblage du fichier binaire plutôt qu'une décompilation. Le désassemblage est une méthode permettant d'obtenir le code assembleur (dans notre cas, le code assembleur x86) d'un programme à partir de son exécutable binaire [21].

Plusieurs outils existent afin d'effectuer un désassemblage de fichiers exécutables. Parmis ces outils, nous avons choisi de travailler avec l'outil GNU *OBJDUMP* car ce dernier propose plusieurs options qui nous seront extrêmement intéressantes par la suite.

3.1.5 Convention d'écriture Intel

Comme expliqué dans l'introduction, il existe deux conventions d'écriture concernant le code machine x86 : Intel et AT&T. Tout au long de nos recherches, nous avons exclusivement travaillé avec la syntaxe Intel et c'est donc celle-ci que nous allons utiliser pour la rédaction de ce document.

Hormis le fait que la syntaxe Intel fût celle que nous trouvions la plus compréhensible, concise et facile d'utilisation, la principale raison de notre choix s'explique par le fait que, contrairement à la syntaxe AT&T, chaque instruction x86 ne peut s'écrire que d'une seule et même façon. En effet, nous avons expliqué dans l'introduction que la convention d'écriture AT&T ajoutait un suffixe aux opérateurs en fonction de la taille de ses opérandes.

Or, dans notre logique de traduction séquentielle des instructions x86 en clauses de Horn, cela nous rajoute des instructions à gérer (car il y a plusieurs opérateurs différents) contrairement à la syntaxe Intel, comme nous l'illustre la comparaison ci-dessous. Autrement dit, cela nous évite de devoir passer du temps à regrouper différentes instructions qui réalise la même opération, comme une addition par exemple.

AT&T	Intel
mov al,bl	movb %bl,%al
mov ax,bx	movw %bx,%ax
mov eax,ebx	movl %ebx,%eax

FIGURE 3.2 – Pattern d'une instruction x86 renvoyée par *OBJDUMP*

3.2 Traduction depuis l'hexadécimal ou du code machine

Si nous reprenons le désassemblage de la section *.text* que nous avons réalisé dans la section précédente, nous pouvons remarquer que le résultat ne ressemble pas tout à fait à du code assembleur. En effet, les instructions du code assembleur x86 sont toutes précédées d'une série de nombre hexadécimaux. Il s'agit, en réalité, de l'adresse mémoire et de l'équivalent en hexadécimal de l'instruction, comme illustré sur la figure 3.2.

L'adresse mémoire des instructions nous importe peu pour le moment mais leur équivalent hexadécimal est particulièrement intéressant. En effet, étant donné que nous devons traduire ces instructions assembleur en clauses de Horn, nous avons donc deux possibilités.

1. Traduire les instructions x86 sous forme de code assembleur
2. Traduire les instructions x86 sous forme de code machine

Avant d'aller plus loin, revenons brièvement sur les notions de code machine et code assembleur.

Le code (ou langage) machine représente le langage directement interprétable par le processeur, c'est-à-dire du code binaire (ici, traduit en hexadécimal), tandis que le code (ou langage) assembleur est une représentation humainement plus compréhensible et lisible du code machine.

Cela se fait via l'utilisation des codes opérateurs (*opcode*). Chaque instruction possède un "numéro d'instruction" (*opcode*), qui est exprimé par défaut en hexadécimal, et c'est à partir de ce dernier qu'est généré le code machine de l'instruction en question. Par exemple, l'opcode *6A* correspond à l'instruction *push* et de ce fait, l'instruction *push 5* s'écrira donc *6A 0x5* et sera donc traduite en code machine de la manière suivante *01101010 0101*. Ainsi, il serait intéressant de se demander si les *opcodes* permettrait une traduction des instructions en clauses de Horn plus simple qu'en se basant sur le code assembleur x86.

La réponse à cette question est assez rapide à trouver car il s'avère que l'*opcode* d'une instruction varie en fonction de ses opérandes. En effet, pour reprendre l'exemple précédent, l'*opcode 6A* correspond à l'instruction *PUSH* mais uniquement lorsque celle-ci est suivie d'une valeur immédiate de 8 bits. L'*opcode* de l'instruction *PUSH* suivie d'un registre de 32 bits (comme *PUSH EAX*) n'est

donc plus *6A* mais *50*[1].

Par conséquent, nous nous apercevons rapidement que, à l'instar de la raison pour laquelle nous avons préféré la convention d'écriture Intel à AT&T, nous nous retrouvons face à plusieurs façon d'écrire une même instruction, ce qui nous complique la tâche quant à notre idée de traduction séquentielle.

3.3 Une méthode de traduction séquentielle

3.3.1 Idée générale

Maintenant que notre environnement de travail est fixé, nous allons introduire et expliquer notre approche quant à la traduction des instructions x86 en clauses CLP. La principale difficulté à laquelle doit faire face une traduction d'un programme est qu'elle doit pouvoir représenter de façon fidèle l'état actuel de la machine à n'importe quel instant.

Pour ce faire, nous avons opté pour une traduction que nous pouvons qualifier de séquentielle. En effet, une traduction séquentielle nous semble être une bonne approche car le code assembleur x86 se lit et s'exécute de façon séquentielle, c'est-à-dire une instruction à la fois. Voici les points clés de la méthode de traduction que nous proposons.

1. Chaque instruction x86 est traduite par une (ou plusieurs) clause(s) CLP qui tradui(sen)t son action.
2. L'état de la machine à un instant q (à l'instruction q) est représenté par les arguments de la clause CLP q correspondant à l'instruction q.
3. Le passage d'une instruction q à l'instruction $q+1$, et donc le passage d'un état q à l'état $q+1$, est traduit par l'appel de la clause CLP $q+1$ par la clause CLP q.

Cette façon de faire implique de devoir définir, pour chaque instruction existante dans l'ensemble des instructions x86, son équivalent en clauses CLP. Il va de soi qu'il nous est impossible, dans le temps qu'il nous est imparti, de traiter l'entièreté de cet ensemble et c'est pourquoi nous allons, dans le cadre de nos recherches, nous focaliser sur les instructions les plus communes comme les instructions arithmétiques, l'utilisation de la pile,...

3.3.2 Modèle de traduction

Avant même de traduire des instructions simples, il convient de définir un *pattern* de clause en Horn pour nos traductions. Ce *pattern* doit permettre à une clause CLP de traduire l'action d'une instruction x86 (une addition, par exemple), d'appeler la clause CLP correspondant à l'instruction suivante et surtout, de représenter l'état de la machine à ce moment-mà.

1. Plus précisément, $50 + r$ où r est le code registre du registre

Comme l'un de nos principaux objectifs de ce projet est de réaliser un outils qui serait capable de traduire automatiquement un fichier binaire exécutable en un équivalent en clauses de Horn et dont ce résultat pourrait directement être analysé par d'autres outils (comme [26] pour une comparaison de programme), nous avons décidé de représenter nos clauses de Horn sous une forme de clauses CLP selon une syntaxe conceptuelle qui se rapproche fortement de la syntaxe Prolog, comme expliqué dans la section 1.2. Par abus de langage, nous nous autorisons l'usage du terme "clause Prolog avec contraintes. Voici un exemple de clauses CLP avec lesquelles nous travaillons.

Exemple Une clause CLP.

predicat1(V1,V2,V3) :- {V2>V1}, predicat2(V1,V2,V3), predicat3(V2,V3).

Représentation de l'état de la machine L'état de la machine est représenté par la valeur de ses différents registres et par les valeurs présentes sur la pile, il nous faut donc pouvoir représenter ces derniers dans nos clauses CLP. L'interaction avec la pile étant particulièrement plus compliquée qu'avec les registres, nous allons dans un premier temps nous intéresser à la représentation des registres et nous reviendrons sur la représentation de la pile dans la section 4.2.

Les registres de travail généraux étant au nombre de huit [2], nous proposons de les représenter en clause Prolog avec contrainte *via* les arguments de celles-ci. Ainsi, chaque registre serait représenté par un argument spécifique et ceux-ci seront donc omniprésents au sein de chaque clause CLP.

Représentation de l'opération des instructions Comme dit précédemment, notre méthode implique de devoir concevoir une traduction CLP pour chaque instruction x86. Afin de réaliser cela, nous verrons dans la section suivante que nous tirons profit de la possibilité d'émettre des contraintes offerte par la programmation logique par contraintes.

Représentation du passage d'un état q à l'état $q+1$ Afin de transférer le contrôle depuis la clause CLP représentant l'instruction q à la clause CLP représentant l'instruction suivante, tout en veillant à conserver l'état actuel de la machine à l'instant q, nous appelons simplement le prédicat composant la tête de la clause représentant l'instruction $q+1$.

Afin de représenter nos traductions d'instructions x86 en clauses CLP, nous allons utiliser la notation suivante :

$$\frac{Instruction\ x86}{Clause\ CLP,\ constituée\ d'une\ tête\ et\ d'un\ corps,\ équivalente}$$

Exemple Traduction de l'instruction *ADD EAX, 3* en une clause CLP selon notre méthode.

$$\frac{add\ EAX, 3}{p_q(EAX,EBX,ECX,EDX,ESI,EDI,EBP,ESP) \quad :- \quad p_{q+1}(EAX{+}3,EBX,ECX,EDX,ESI,EDI,EBP,ESP).}$$

2. Car nous travaillons dans une architecture 32 bits

A travers ce simple exemple, nous pouvons constater que l'addition est bien prise en compte, pour peu que les prédicats suivants respectent tous le même nombre et le même ordre des arguments, ce qui est le cas. Intuitivement, cela laisse penser que nous pouvons aisément appliquer cette technique aux trois autres opérations arithmétiques, à savoir la soustraction, la multiplication et la division. C'est ce que nous allons voir dans la section suivante. Enfin, nous nous aperçevons que, pour ces opérations arithmétiques, la programmation logique par contraintes n'a pas grand intérêt. Nous allons également voir dans quels cas celle-ci nous sera utile.

Chapitre 4

Application du modèle de traduction

Dans ce chapitre, nous allons appliquer la méthode définie dans la section précédente (notre *pattern* de traduction) sur un ensemble d'instructions x86 relativement basiques, à savoir des instructions arithmétiques, des instructions de branchement et des instructions de déplacement de donnée. Dans les traductions suivantes, nous ne représenterons pas les registres *ESI* et *EDI* car ceux-ci ne sont presque jamais utilisés par les compilateurs et cela nous permet d'épurer et de rendre plus lisibles nos clauses CLP. Notons que ces deux registres sont toutefois bien pris en compte dans l'outil développé en parallèle de ces recherches.

4.1 Instruction basiques

4.1.1 Instructions arithmétiques

Les opérations d'additation, de soustraction, de multiplication et de division sont respectivement représentées par les opérateurs *ADD, SUB, IMUL, IDIV* en code assembleur x86. Comme nous l'avons dit précédemment, intuitivement, la méthode appliquée sur l'exemple d'addition vu dans la section précédente peut également s'appliquer sur ces opérations arithmétiques mais nous allons voir que la division demande une attention particulière.

Addition

$$\frac{add\,EAX,3}{p_q(EAX,EBX,ECX,EDX,EBP,ESP) \quad :- \quad p_{q+1}(EAX{+}3,EBX,ECX,EDX,EBP,ESP).}$$

Soustraction

$$\frac{add\,EBX,3}{p_q(EAX,EBX,ECX,EDX,EBP,ESP) \quad :- \quad p_{q+1}(EAX,EBX{-}3,ECX,EDX,EBP,ESP).}$$

Multiplication La multiplication en code assembleur x86 peut prendre deux formes différentes [1], chacune ayant sa spécificité. La première forme prend deux opérandes et effectue une multiplication entre ces deux opérandes et place le résultat dans le premier opérande et la deuxième forme prend trois opérandes, effectue une multiplication des deux derniers et place le résultat dans le premier opérande.

$$\frac{imul\ EBX, 5}{p_q(EAX,EBX,ECX,EDX,EBP,ESP) \quad :- \quad p_{q+1}(EAX,EBX*5,ECX,EDX,EBP,ESP).}$$

$$\frac{imul\ EBX, EAX, 5}{p_q(EAX,EBX,ECX,EDX,EBP,ESP) \quad :- \quad p_{q+1}(EAX,EAX*5,ECX,EDX,EBP,ESP).}$$

Division La division, quant à elle, ne peut pas se traduire aussi simplement car les opérations qu'elle effectue sont plus complexes que pour les opérations précédentes. En effet, la division ne prend qu'un seul opérande et effectue une division de la concaténation du registre *EDX* avec le registre *EAX* (*EDX* représentant les bits de poids fort) par cet opérande. Le résultat de cet division est ensuite renvoyé tel que *EAX* contienne le quotient et que *EDX* contienne le reste.

La concaténation binaire de deux nombre n'étant pas une opération gérée par la programmation logique par contraintes, nous devons donc la traduire la traduire en opérations arithmétiques décimales. Cela signifie qu'il nous faut trouver un moyen d'obtenir le nombre décimal émanant de cette concaténation à partir des deux nombres initiaux (*EDX* et *EAX*). Pour ce faire, décomposons la concaténation binaire de *EDX* et *EAX* en plusieurs étapes atomiques afin de les traduire une à une :

1. *EDX* et *EAX* étant en 32 bits, la concaténation des deux implique un *shift-left* [2] de 32 bits du registre qui représente les bits de poids fort, c'est-à-dire *EDX*. Autrement, cela implique d'ajouter 32 bits de poids faible initialisés à 0.
2. Ensuite, il suffit de remplacer les 32 nouveaux bits de poids faible par le registre *EAX* afin d'obtenir une contaténation finale de 64 bits.

Pour la première opération, nous savons qu'effectuer un *shift-left* de X sur un nombre binaire revient, en décimal, à multiplier ce nombre par 2^X. Ensuite, comme le registre *EAX* vient prendre la place des 32 bits de poids faible du nombre obtenu après le *shift-left*, cela revient tout simplement, à additionner ce nombre. Vérifions nos dires sur un simple exemple.

Exemple Soient le nombre binaire 1100 correspondant à 12 en décimal et le nombre binaire 1001 correspondant à 9 en décimal. La concaténation de ces deux nombre de quatre bits telle que 1100 1001 équivaut à 201 en décimal. Vérifions

1. En réalité, il existe une troisième forme peu souvent utilisée mais nous y reviendrons plus tard car celle-ci nécessite des notions plus avancées
2. Déplacement des bits vers la gauche

que nous pouvons retrouver le nombre 201 à partir des nombres 9 et 12.

Le nombre 12 représentant les quatre bits de poids fort, nous effectuons d'abord un *shift-left* de 4 bits, c'est-à-dire une multiplication de 12 par 2^4. Nous obtenons donc le résultat de 192. Il ne nous reste plus qu'à additionner le nombre représetnant les 4 bits de poids faible, c'est-à-dire 9, et nous obtenons bien le résultat de 201.

Nous sommes maintenant capable de traduire la division en clauses CLP. Notons que, dans notre notation, les opérateurs "/" et "//" représentent respectivement la division entière et le modulo.

$$\frac{idiv\ EBX}{p_q(EAX,EBX,ECX,EDX,EBP,ESP) \quad :- \quad p_{q+1}((EDX*2^{32}+EAX)/EBX,EBX,ECX,(EDX*2^{32}+EAX)//EBX,EBP,ESP).}$$

Nous reviendrons sur cet aspect de traduction d'opérations binaires dans la section 6.2.

Incrémentation et décrémentation L'incrémentation et la décrémentation sont représentées par les opérateurs *inc* et *dec*. Il s'agit tout simplement d'un addition et d'une soustraction de 1.

$$\frac{inc\ EAX}{p_q(EAX,EBX,ECX,EDX,EBP,ESP) \quad :- \quad p_{q+1}(EAX+1,EBX,ECX,EDX,EBP,ESP).}$$

$$\frac{dec\ EAX}{p_q(EAX,EBX,ECX,EDX,EBP,ESP) \quad :- \quad p_{q+1}(EAX-1,EBX,ECX,EDX,EBP,ESP).}$$

Priorité des opérateurs Jusqu'ici nous avons travaillé avec des opérandes atomiques, c'est-à-dire ne contenant q'un seul élement cependant, le code assembleur x86 permet de réaliser des opérations arithmétiques au sein d'un même opérande comme par exemple l'instruction suivante : *IMUL EAX, 3+5*2*. Si l'on s'en tient à notre méthode actuelle, nous obtiendrons le résultat suivant.

$$\frac{imul\ EAX, 3+5*2}{p_q(EAX,EBX,ECX,EDX,...) \quad :- \quad p_{q+1}(EAX*3+5*2,EBX,ECX,EDX,...).}$$

Nous pouvons clairement constater que le résultat obtenu *via* notre traduction actuelle n'est pas équivalent au résultat escompté. Soit EAX qui équivaut 3, l'instruction x86 renvoie $EAX = 3 * (3 + 2 * 5) = 39$ et notre résultat CLP renvoie $EAX = 3 * 3 + 5 * 2 = 19$.

Afin de résoudre ce problème, nous proposons d'ajouter des paranthèses sur chaque opérande afin de conserver la priorité des opérateurs des instructions x86. Ainsi nous obtiendrons désormais le résultat suivant.

$$\frac{imul\ EAX, 3+5*2}{p_q(EAX,EBX,ECX,EDX,...) \quad :- \quad p_{q+1}((EAX)*(3+5*2),EBX,ECX,EDX,...).}$$

4.1.2 Instructions de branchement

Les instructions de branchement permettent de rediriger le flux d'exécution du programme à certains endroits du code. Nous avons vu dans la section 3.2 que chaque instruction x86 était précédée par une adresse mémoire, celles-ci servent, entre autre, à pouvoir rediriger le flux d'exécution du programme vers leur instruction, nous parlons dès lors de *labels*. Un *label* est une "étiquette" servant donc à pouvoir rediriger le flux d'exécution du programme vers l'instruction qu'il détermine *via* des instructions de branchement. En code assembleur x86, celui-ci peut être constitué de caractères et de chiffres.

Sauts inconditionnels Le saut inconditionnel est représenté par l'opérateur *JMP*. Ce denier ne prend qu'un seul opérande qui n'est autre que le *label* vers lequel il doit rediriger le flux d'exécution.

Dans la section précédente, nous avons expliqué que notre méthode de traduction impliquait que chaque instruction d'un programme x86 était traduite par une ou plusieurs clauses CLP. Autrement dit, cela implique que chaque instruction x86 possède son propre prédicat au sein du programme CLP (la tête de la clause). De ce constat, nous pouvons nous baser sur l'idée qu'un saut dans le x86 vers une instruction spécifique serait simplement représenté par un appel du prédicat correspondant à cette instruction dans le programme CLP.

$$\frac{jmp\ maFonct}{p_q(EAX,EBX,ECX,EDX,EBP,ESP) \quad :- \quad maFonct(EAX,EBX,ECX,EDX,EBP,ESP).}$$

Sauts conditionnels Comme leur nom l'indique, les sauts conditionnels effectuent la même opération que les sauts inconditionnels mais cette fois-ci, moyennant une condition. Cependant, comme le code assembleur x86 est bas niveau et séquentiel, il n'effectue qu'une opération à la fois et ne peut donc pas effectuer une évaluation (la condition) et un branchement. C'est pourquoi les instructions de saut conditionnel seront toujours précédées d'une instruction permettant d'effectuer l'évaluation (la condition), cette instruction est représentée par l'opérateur *CMP*.

L'instruction *CMP* effectue une comparaison entre deux arguments, par exemple *CMP EAX, EBX* va comparer la valeur des deux registres en effectuant une soustraction de l'un à l'autre. En fonction du résultat obtenu, l'instruction va modifier certains bits spécifiques du registre *eflags* que nous avons vu dans la section 2.2. C'est donc en regardant les bits spécifiques aux opérations arithmétiques du registre *eflags* que nous pouvons savoir si *EAX* est supérieur, égal ou inférieur à *EBX*.

Ainsi, pour traduire des instructions de saut conditionnel, nous devons traduire une comparaison et un branchement à la manière d'un *IF-THEN-ELSE*.

Operateur x86	Description	Contrainte IF-THEN	Contrainte ELSE
je EAX, EBX	jump if equal	EAX = EBX	EAX <> EBX
jne EAX, EBX	jump if not equal	EAX <> EBX	EAX = EBX
jg EAX, EBX	jump if greater	EAX > EBX	EAX =< EBX
jge EAX, EBX	jump if greater or equal	EAX >= EBX	EAX < EBX
jl EAX, EBX	jump if less	EAX < EBX	EAX >= EBX
jle EAX, EBX	jump if less or equal	EAX =< EBX	EAX > EBX

TABLE 4.1 – Opérateurs de sauts conditionnels et leurs contraintes CLP

Voici un exemple de saut conditionnel (si *EAX* est plus grand ou égal à *EBX*) en code assembleur x86.

```
cmp eax, ebx
jge maFunct
...
maFunct :
...
```

Voici la traduction de ce saut conditionnel en clauses CLP.

```
p1(EAX,EBX,ECX,EDX,EBP,ESP) :- {EAX >= EBX},
                               maFunct(EAX,EBX,ECX,EDX,EBP,ESP).
p1(EAX,EBX,ECX,EDX,EBP,ESP) :- {EAX < EBX},
                               p2(EAX,EBX,ECX,EDX,EBP,ESP).
p2(EAX,EBX,ECX,EDX,EBP,ESP) :- ...
...
maFunct(EAX,EBX,ECX,EDX,EBP,ESP) :- ...
```

A travers cet exemple, nous pouvons nous aperçevoir de l'utilité que nous apporte la programmation logique avec contraintes. En effet, le fait de pouvoir utiliser des contraintes nous permet de traduire l'aspect conditionnel de ce type d'instruction de branchement. Nous voyons bien que le prédicat *p1* traduit le saut conditionnel et permet de rediriger le flux d'exécution soit vers le prédicat *maFonct* qui traduit la première instruction de la fonction *maFonct*, soit vers le prédicat *p2* qui traduit l'instruction suivant l'instruction *JGE maFonct*.

Cette façon de faire peut s'appliquer à toutes les autres instructions de saut conditionnels que nous reprenons dans le tableau 4.1 avec leurs contraintes CLP respectives.

4.1.3 Instruction de déplacement de donnée

Affectation L'affectation peut être assimilée à l'opérateur *MOV* en x86, il consiste à copier la valeur du second opérande dans le premier. La traduction de cette instruction se fait tout simplement de la manière suivante.

$$\frac{mov\ EAX, EBX}{p_q(EAX,EBX,ECX,EDX,EBP,ESP) \quad :- \quad p_{q+1}(EBX,EBX,ECX,EDX,EBP,ESP).}$$

Interactions avec la pile Comme nous l'avons expliqué, il est également possible de placer et déplacer des données sur la pile via les instructions *PUSH* et *POP*. Nous reviendrons sur celles-ci dans la sous-section 4.2.2 où nous expliquons comment nous représentons la pile en clauses CLP.

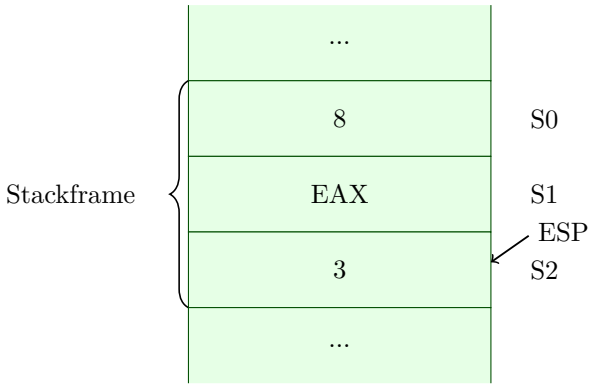
4.2 La pile x86

Maintenant que nous savons traduire une bonne partie des instructions les plus communes du code assembleur x86, il nous faut maintenant pouvoir représenter la pile car celle-ci est utilisée en permanence dans les programme x86 et qu'elle est indispensable pour représenter l'état de la machine. Dans cette section, nous allons voir comment nous représentons cette pile en CLP et comment nous traduisons les instructions l'affectant.

4.2.1 Idée générale

Contrairement aux registres où nous avons un nombre fixe qui ne change jamais, la pile quant à elle possède un nombre d'éléments indéterminé et qui varie en permanence lors de l'exécution d'un programme. De ce fait, la méthode de représentation que nous proposons pour la pile x86 est de représenter tous ses éléments de la même manière que les registres, c'est-à-dire *via* les arguments des clauses CLP mais comme le respect du même ordre d'argument entre les clauses CLP est primordial et que nous connaissons le nombre fixe de registres, à savoir huit, nous proposons de placer les arguments représentant les éléments de la pile après ceux qui représentent les huit registres. Ainsi, le premier élément empilé sur la pile sera donc le $9^{\text{ème}}$ argument des clauses CLP, le second le $10^{\text{ème}}$ et ainsi de suite.

Exemple Afin d'illustrer nos propos, voici la clause CLP que nous aurons pour une pile telle qu'affichée ci-dessous.



Clause CLP représentant l'état de la machine à cet instant où *S0, S1, S2* représentent respectivement les valeurs *8, EAX, 3*.

predicat(EAX,EBX,ECX,EDX,EBP,ESP,S0,S1,S2) :- ...

4.2.2 Traduction des instructions

Comme nous le savons, l'empilage et le désempilage sont respectivement représentés par les opérateurs *PUSH* et *POP*. *PUSH* copie la valeur de son opérande au sommet de pile et *POP* déplace (et non pas copie) la valeur se trouvant au sommet de la pile dans son opérande.

$$\frac{push\ X}{p_q(EAX,EBX,ECX,EDX,EBP,ESP,S0,S1) \quad :- \quad p_{q+1}(EAX,EBX,ECX,EDX,EBP,ESP,S0,S1,X).}$$

$$\frac{pop\ EAX}{p_q(EAX,EBX,ECX,EDX,EBP,ESP,S0,S1) \quad :- \quad p_{q+1}(S1,EBX,ECX,EDX,EBP,ESP,S0).}$$

Nous avons également vu que pour supprimer des éléments se trouvant au sommet de la pile, nous pouvions également effectuer une addition sur le registre *ESP*, qui pointe vers le sommet de *stackframe* courant. Une soustraction sur *ESP* permet, à l'inverse, de réserver des *slots* sur la pile. Nous proposons les traductions suivantes.

$$\frac{add\ ESP, 8}{p_q(...,S0,S1,S2) \quad :- \quad p_{q+1}(...,S0).}$$

$$\frac{sub\ ESP, 8}{p_q(...,S0) \quad :- \quad p_{q+1}(...,S0,NULL,NULL).}$$

Enfin, nous sommes maintenant capables de traduire les instructions de la section précédente manipulant les éléments contenus dans la pile comme le montre les deux exemples ci-dessous.

$$\frac{mov\ EAX, [EBP-8]}{p_q(EAX,EBX,ECX,EDX,EBP,ESP,S0,S1,S2) \quad :- \quad p_{q+1}(S1,EBX,ECX,EDX,EBP,ESP,S0,S1,S2).}$$

$$\frac{add\ EAX, [ESP]}{p_q(EAX,EBX,ECX,EDX,EBP,ESP,S0,S1,S2) \quad :- \quad p_{q+1}(EAX+S2,EBX,ECX,EDX,EBP,ESP,S0,S1,S2).}$$

4.3 L'appel de fonction

Dans la section précédente, nous nous sommes penchés sur la représentation de la pile et nous avons vu que les instructions basiques étaient correctement traduites. Cependant, l'un des principaux usages de la pile est le passage de paramètre lors de l'appel de fonction. Nous allons donc, dans cette section, représenter en clauses CLP un appel de fonction ainsi que l'exécution de celle-ci.

Un appel de fonction implique plusieurs points importants affectant à l'état de la machine.

1. Passage des arguments par la pile
2. Création d'un nouveau *stackframe* et transfert du contrôle du flux d'éxecution
3. Suppression du stackframe et retransfert du contrôle du flux d'exécution

La passage des arguments par la pile ne pose pas de problème car ceci se fait *via* l'utilisation d'un *PUSH* ou d'un *MOV* par rapport à *EBP*. Cependant, nous avons travaillé jusqu'ici avec (au sein de) un seul *stackframe* et cela est maintenant problématique. Reprenons l'exemple vu dans la section 2.4 afin d'illustrer les problèmes auxquels nous sommes confrontés et la solution que nous proposons.

Exemple Soient P un programme et *maFonct* une fonction appelée par P qui calcule la somme des deux paramètres reçus en entrée.

Code C du programme$P$:

```
int main() {
int a = 3;
int b = 2;
return maFonct(a,b);
}

int maFonct(int x, int y) {
return x + y;
}
```

Code machine x86 du programme P :

```
...
push 0x2 ; Push du deuxième paramètre (2), écrit en hexadécimal
push 0x3 ; Push du premier paramètre (3), écrit en hexadécimal
call maFonct ; Appel de la maFonct
add esp, 0x8 ; Suppression des paramètres précédemment empilés
...

maFonct: ; Label qui détermine la fonction "maFonct"
```

```
push ebp ; sauvegarde de EBP
mov ebp, esp ; définition de la base du stackframe de maFonct

sub esp, 4 ; réservation de l'espace pour la variable locale
mov eax, [ebp+12] ; eax = 3
mov edx, [ebp+8] ; edx = 2
mov [ebp-4], eax ; variable locale = eax
add [ebp-4], edx ; variable locale = variable locale + edx
mov eax, [ebp-4] ; résultat final placé dans eax

mov esp, ebp ; suppression du stackframe de maFonction
pop ebp ; restauration de l'ancien EBP
ret ; retour à l'appelant
```

Avant de continuer, il nous reste deux instructions à traduire en clauses CLP. En effet, nous avons vu que pour appeler une fonction, l'appelant utilise la fonction *CALL* et l'appelé, après s'être exécuté, renvoie le flux d'exécution à l'appelant *via* l'instruction *RET*.

Dans la section 2.4, nous avons vu que *CALL* effectuait un empilement du registre *EIP* sur la pile avant de transférer le contrôle à l'appelé et que *RET* effectuait un désempilement de cette valeur avant de retransférer le contrôle à l'appelant. Dans notre logique de traduction, cela donnerait donc ceci.

$$\frac{call\ LABEL}{p_q(...,S0,S1,S2) \quad :- \quad label_0(...,S0,S1,S2,EIP).}$$

$$\frac{ret}{lavel_q(S0,S1,S2,S3) \quad :- \quad p_{q+1}(...,S0,S1,S2).}$$

Précisons que *label0* correspond à la première instruction de la fonction appelée.

4.3.1 Problèmes

Comme déjà dit, un *stackframe* est défini par *EBP* qui en représente la base et *ESP* qui en représente le sommet, or jusqu'ici, nous avons travaillé avec un seul *stackframe*, c'est-à-dire le *stackframe* courant. Cela implique, par définition, qu'il nous est impossible d'accéder à des valeurs qui se trouveraient au-dessus [3] de *EBP* (dans un autre *stackframe*) et, en effet, nous avons défini dans la section 2.4 que *EBP* pointait vers les $9^{\text{ème}}$ argument de nos clauses CLP, à savoir le premier élément de la pile.

Le premier problème auquel nous sommes donc confrontés est que lorsque la fonction appelée s'exécute, elle va trouver ses paramètres par rapport à *EBP* mais dans le *stackframe* de l'appelant et non pas dans le sien. Cela se démontre par les instructions *mov eax, [ebp+12]* et *mov edx, [ebp+8]* que nous apercevons dans notre exemple.

3. Rappelons que la pile croît vers le bas

Le second problème auquel nous sommes confrontés vient de fait que lorsque nous traduisons une instruction *CALL*, nous ne prenons pas en compte l'instruction suivante. En effet, la traduction que nous avons proposée jusqu'ici traduit bien le transfert du flux d'exécution vers la fonction appelée mais lorsque cette dernière aura fini de s'exécuter, nous nous retrouverons face à une instruction *RET* et nous ne serons pas capables de savoir vers quelle clause CLP rediriger le flux d'exécution, à savoir la clause CLP représentant l'instruction suivant l'instruction *CALL*.

4.3.2 Solution

La solution que nous proposons pour répondre à ces problèmes est d'imiter au maximum le comportement d'un appel de fonction dans un langage de programmation haut niveau. Plus précisément, nous proposons que, lorsque nous traduisons une fonction, seuls les registres de travail généraux et les paramètres de cette fonction soient représentés dans les clauses CLP traduisant cette fonction.

Cette manière de faire revient donc à ne plus imbriquer les *stackframes* entre eux comme c'est le cas sur la pile x86 mais à diviser le programme en fonction de ses *stackframes*. Ainsi, les clauses CLP traduisant la fonction *main* seront uniquement constituées des registres de travail généraux car nous partons de l'hypothèse que la fonction *main* ne prend pas de paramètres en entrée et les clauses CLP traduisant les autres fonctions présentes dans le programme X86 seront, quant à elles, constituées des registres de travail généraux et des paramètres, empilés sur la pile, nécessaires à leur bonne exécution.

Avant de mettre en œuvre cette solution, celle-ci engendre deux questions auxquelles nous devons répondre.

1. Comment déterminer le nombre de paramètres nécessaires pour une fonction donnée et lesquels ?
2. Maintenant que le programme est divisé en plusieurs fonctions indépendantes (division en fonction des *stackframe*), comment représenter le transfert du résultat de la fonction appelée à la fonction appelante ?

Pour répondre à la première question, nous avons vu que pour appeler une fonction, le programme x86 devait, avant l'appel de celle-ci, empiler les paramètres *input* sur la pile et une fois que la fonction s'est exécutée, et donc après son appel, le programme devait supprimer ces paramètres *via* l'instruction *ADD ESP, X*4*, où X est le nombre de paramètres. Cela signifie que pour déterminer les paramètres nécessaires à une fonction, il nous suffit de reprendre les X derniers éléments empilés sur la pile avant l'appel de cette fonction. Par exemple, si l'instruction suivant l'appel d'une fonction est *ADD ESP, 8*, cela signifie que la fonction en question demande deux paramètres en *input*.

Ensuite, pour répondre à la deuxième question, nous proposons de rajouter un argument R dans nos clauses CLP afin de représenter le résultat de la fonction traduite. Cela est applicable car le programme principale, c'est-à-dire le *main*, étant lui-même une fonction, il revoie également un résultat (en C, le résultat pour annoncer que tout s'est déroulé correctement est 0). Ainsi, tous nos

ensembles de clauses CLP représentant chacun une fonction comporteront dans leurs arguments, un argument R qui contiendra le résultat final de la fonction traduite. Cet argument ne sera donc utilisé qu'en fin de fonction pour recevoir le résultat.

Enfin, afin de transmettre le résultat de la fonction appelée à la fonction appelante, nous proposons de modifier nos traductions précédentes des instructions *CALL* et *RET* de la manière suivante.

Soit une fonction nécessitant deux paramètres traduite par un ensemble de clauses CLP *labelX* où X est un entier positif.

$$\frac{ret}{label_q(...,R,S0,S1) \quad :- \quad \{R{=}EAX\}.}$$

$$\frac{call\ LABEL}{p_q(...,R,S0,S1,S2) \quad :- \quad label_0(...,TMP,S1,S2), p_{q+1}(TMP,EBX,ECX,EDX,EBP,ESP,R,S0,S1,S2).}$$

Notons que l'argument R représente le résultat de la fonction appelée et n'est donc pas à confondre avec le résultat de la fonction appelante, qui est R. Ensuite, le résultat *R2* est assigné au registre *EAX* comme le stipule la convention d'appel de fonction vu précédemment.

Reprenons donc notre exemple de début de section et voyons les clauses CLP produites suite à notre méthode.

```
p_start(R)
:- p0(EAX,EBX,ECX,EDX,EBP ESP,R).

p0(EAX,EBX,ECX,EDX,EBP,ESP,R)
:- p1(EAX,EBX,ECX,EDX,EBP,ESP,R,2) % push 2.

p1(EAX,EBX,ECX,EDX,EBP,ESP,R,S0)
:- p2(EAX,EBX,ECX,EDX,EBP,ESP,R,S0,3) % push 3.

p2(EAX,EBX,ECX,EDX,EBP,ESP,R,S0,S1)
:- maFunct0(EAX,EBX,ECX,EDX,EBP,ESP,R2,S0,S1),
   p3(R2,EBX,ECX,EDX,EBP,ESP,R,S0,S1). % call maFunct

p3(EAX,EBX,ECX,EDX,EBP,ESP,R,S0,S1)
:- p4(EAX,EBX,ECX,EDX,EBP,ESP). % add esp, 8

...

maFunct0(EAX,EBX,ECX,EDX,EBP,ESP,R,S0,S1)
:- maFunct1(EAX,EBX,ECX,EDX,EBP,ESP,S0,S1,EBP). % push ebp

maFunct1(EAX,EBX,ECX,EDX,EBP,ESP,R,S0,S1,S2)
:- {EBP=ESP}, maFunct2(EAX,EBX,ECX,EDX,EBP,ESP,R,S0,S1,S2). % mov ebp, esp

maFunct2(EAX,EBX,ECX,EDX,EBP,ESP,R,S0,S1,S2)
:- maFunct3(EAX,EBX,ECX,EDX,EBP,ESP,R,S0,S1,S2,NULL). % sub esp, 4
```

```
maFunct3(EAX,EBX,ECX,EDX,EBP,ESP,R,S0,S1,S2,S3)
:- {EAX=S0}, maFunct4(EAX,EBX,ECX,EDX,EBP,ESP,R,S0,S1,S2,S3).
% mov eax, [ebp+12]

maFunct4(EAX,EBX,ECX,EDX,EBP,ESP,R,S0,S1,S2,S3)
:- {EDX=S1}, maFunct5(EAX,EBX,ECX,EDX,EBP,ESP,R,S0,S1,S2,S3).
% mov eax, [ebp+8]

maFunct5(EAX,EBX,ECX,EDX,EBP,ESP,R,S0,S1,S2,S3)
:- {S3=EAX}, maFunct6(EAX,EBX,ECX,EDX,EBP,ESP,R,S0,S1,S2,S3).
% mov [ebp-4], eax

maFunct6(EAX,EBX,ECX,EDX,EBP,ESP,R,S0,S1,S2,S3)
:- {S3=S3+EAX}, maFunct7(EAX,EBX,ECX,EDX,EBP,ESP,R,S0,S1,S2,S3).
% add [ebp-4], edx

maFunct7(EAX,EBX,ECX,EDX,EBP,ESP,R,S0,S1,S2,S3)
:- {EAX=S3}, maFunct8(EAX,EBX,ECX,EDX,EBP,ESP,R,S0,S1,S2,S3).
% mov eax, [ebp-4]

maFunct8(EAX,EBX,ECX,EDX,EBP,ESP,R,S0,S1,S2,S3)
:- {EBP=ESP}, maFunct9(EAX,EBX,ECX,EDX,EBP,ESP,R,S0,S1,S2).
% mov esp, ebp

maFunct9(EAX,EBX,ECX,EDX,EBP,ESP,R,S0,S1,S2)
:- {EBP=S2}, maFunct10(EAX,EBX,ECX,EDX,EBP,ESP,R,S0,S1).
% pop ebp

maFunct10(EAX,EBX,ECX,EDX,EBP,ESP,R,S0,S1) :- {R=EAX}.
```

4.4 Les variables globales et constantes

Les variables globales et les constantes font partie intégrante des programmes conçus par les développeurs, il serait donc intéressant de les intégrer dans notre étude. Pour ce faire, nous avons vu dans la section 3.1 que les variables globales initialisées, non-initialisées et les constantes d'un programme exécutable x86 au format ELF se trouvaient dans des sections spécifiques : *.data, .rodata* et *.bss*.

4.4.1 Analyse

Voici un programme C sur lequel nous allons nous baser afin d'illustrer la manière de représenter l'utilisation de ces variables et constantes en clauses CLP que nous proposons.

```
int varG = 256;
int varGni;
const int maConst = 32;

int main() {
    varGni = 5;
    return (varG + maConst)*varGni;
}
```

Dans cet exemple, nous pouvons constater qu'une variable globale *varG* est initialisée à 256, qu'une variable globale *varGni* n'est pas initialisée et qu'une constante *maConst* est déclarée avec une valeur de 16.

Code assembleur x86 du programme :

```
main:
 mov    DWORD PTR ds:0x804a020,0x5
 mov    eax,ds:0x804a018
 mov    edx,0x20
 add    edx,eax
 mov    eax,ds:0x804a020
 imul   eax,edx
 ret
```

Nous pouvons remarquer dans ce code l'utilisation d'adresses mémoire, telle que *ds :0x804a020*, qui nous sont encore inconnues jusqu'ici. Intuitivement, grâce aux instructions, nous pouvons déduire que *ds :0x804a020* correspond à notre variable *varGni*, que *ds :0x804a018* correspond à notre variable *varG* et que notre constante *maConst* n'est pas représentée sous forme d'adresse mémoire mais est directement "traduite" en code assembleur.

Désassemblons maintenant les sections *.data, .rodata* et *.bss* afin de vérifier si nous retrouvons bien ces adresses mémoire.

```
Déassemblage de la section .rodata:

08048478 <_fp_hw>:
 8048478: 03 00 00 00                                        ....

0804847c <_IO_stdin_used>:
 804847c: 01 00 02 00                                        ....

08048480 <maConst>:
 8048480: 20 00 00 00                                         ...

Déassemblage de la section .data:

0804a010 <__data_start>:
 804a010: 00 00                  add    BYTE PTR [eax],al
```

```
...

0804a014 <__dso_handle>:
 804a014: 00 00 00 00                                          ....

0804a018 <varG>:
 804a018: 00 01 00 00                                          ....

Déassemblage de la section .bss:

0804a01c <__bss_start>:
 804a01c: 00 00                  add    BYTE PTR [eax],al
...

0804a020 <varGni>:
 804a020: 00 00 00 00                                          ....
```

Dans le désassemblage de ces sections, nous pouvons constater que nos deux variables globales et notre constante se situent bien aux adresses précédemment citées dans un formatage spécifique en hexadécimal. Notons également que ces sections se trouvent dans le segment *data*, ainsi le diminutif *ds* se trouvant devant les adresses mémoire utilisées dans le programme x86 signifie *data segment*.

4.4.2 Solution

L'approche que nous proposons pour représenter l'utilisation de ce type de variable en clauses CLP est la suivante.

Premièrement, nous avons vu que les constantes étaient directement "traduites" en code assembleur, cela signifie qu'elles ne requièrent pas d'interaction avec des adresses mémoires et que nous pouvons donc les considérer comme des nombres hexadecimaux quelconques.

Quant aux variables globales initialisées et non-initialisées, pour y accéder le système utilise leur adresse mémoire respective. Leur adresse mémoire étant commune pour toutes les fonctions y ayant recourt, nous proposons de représenter ces variables globales par des arguments, de la même façon que les registres de travail généraux (ceux-ci étant également communns à toutes les fonctions d'un programme).

Et enfin, afin de représenter leur initialisation, nous proposons de créer une clause CLP permettant d'assigner les valeurs d'initialisation de ces variables en début de programme, c'est-à-dire avant la première clause CLP représentant la première instruction du programme. Pour les variables globales non-initialisées, nous avons vu qu'elles étaient initialisées à 0 par défaut.

Ainsi, voici le programme CLP que nous obtenons pour l'exemple de cette section en appliquant notre méthode.

```
p_start(R)
:- {V0=256, V1=0}, p1(V0,V1,EAX,EBX,ECX,EDX,EBP,ESP,R).

p1(V0,V1,EAX,EBX,ECX,EDX,EBP,ESP,R)
:- p2(V0,5,EAX,EBX,ECX,EDX,EBP,ESP,R).

p2(V0,V1,EAX,EBX,ECX,EDX,EBP,ESP,R)
:- p3(V0,V1,V0,EBX,ECX,EDX,EBP,ESP,R).

p3(V0,V1,EAX,EBX,ECX,EDX,EBP,ESP,R)
:- p4(V0,V1,EAX,EBX,ECX,32,EBP,ESP,R).

p4(V0,V1,EAX,EBX,ECX,EDX,EBP,ESP,R)
:- p5(V0, V1, EAX, EBX, ECX, EDX + EAX, EBP, ESP, R).

p5(V0,V1,EAX,EBX,ECX,EDX,EBP,ESP,R)
:- p6(V0,V1,V1,EBX,ECX,EDX,EBP,ESP,R).

p6(V0,V1,EAX,EBX,ECX,EDX,EBP,ESP,R)
:- p7(V0, V1, EAX * EDX, EBX, ECX, EDX, EBP, ESP, R).

p7(V0,V1,EAX,EBX,ECX,EDX,EBP,ESP,R)
:- {R = EAX}
```

Chapitre 5

L'outil Hornix

5.1 Hornix, un compilateur CLP

L'un des gros objectifs de ce travail, comme énoncé à quelques reprises, fût de réaliser un outil permettant de mettre en application notre méthode de traduction d'un programme en code assembleur x86 vers un programme CLP. Dans ce chapitre, nous allons donc brièvement passer en revue les points importants de cet outil méritant d'être mentionnés (répartis sous forme de sous-sections), sans rentrer dans les détails car ce n'est pas le but principal de ce mémoire.

Comme l'objectif de cet outil se rapproche fortement de celui d'un compilateur, l'idée générale suivie pour l'implémentation de ce projet était de suivre les principes de conception d'un compilateur. La figure 5.1 illustre l'architecture générale de notre outil.

5.1.1 Désassemblage et traitement de texte

Jusqu'ici nous nous sommes exclusivement intéressés aux instructions x86 et nous avons volontairement évité plusieurs aspect du processus de désassemblage. En effet, lorsque nous désassemblons la section *.text* contenant le code exécutable du fichier ELF, le résultat ne contient pas uniquement la fonction *main* et les fonctions que celle-ci appelle, il contient une série d'autres fonctions qui sont exécutées avant la fonction *main*, certaines servant à initialiser le programme et d'autres étant issues d'optimisations réalisées par le compilateur. À titre d'exemple, l'annexe A représente le désassemblage complet du programme C suivant.

```
int varG = 256;
int varGni;
const int maConst = 32;

int main() {
    varGni = 5;
    return (varG + maConst)*varGni;
}
```

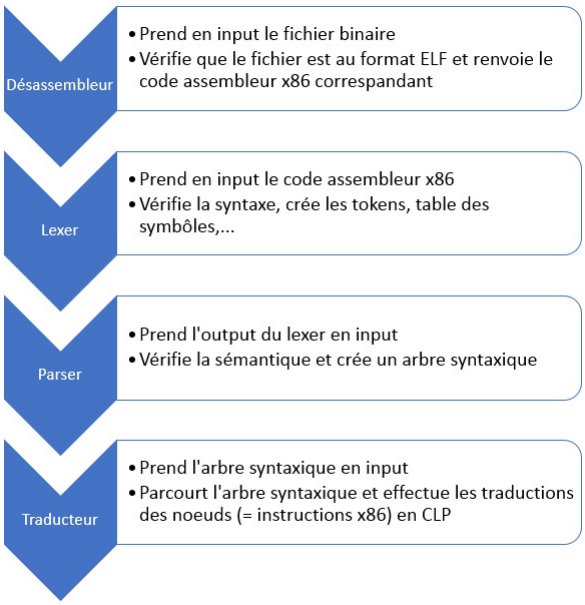


FIGURE 5.1 – Architecture de l'outil Hornix

Un nettoyage de ce désassemblage est donc nécessaire afin de ne garder que les instructions x86 qui nous intéressent. Notre outil prend donc en input un fichier exécutable x86 au format ELF et fait donc appel à l'outil *GNU OBJDUMP* pour désassembler la section *.text* afin d'obtenir le code exécutable du programme ainsi que les sections *.data, .rodata* et *.bss* afin d'obtenir la liste des variables globales et constantes utilisées par le programme et de résoudre les appels *via* leur adresse par la suite.

Suite à cela, l'outil effectue donc un traitement de texte afin de ne garder que les parties de codes qui nous intéressent, à savoir la fonction *main* et les fonctions que celle-ci appelle.

5.1.2 Lexer/Parser

L'une des premières étapes d'un compilateur est l'application d'un *lexer* couplé à un *parser*. Un *lexer* effectue une analyse lexical du code reçu en entrée, c'est-à-dire qu'il analyse le code reçu en entrée caractère par caractère et en renvoie une série de *tokens* qui sont envoyés au *parser*[1].

Un *parser* reçoit donc les *tokens* créés par le *lexer* et vérifie s'il n'y a pas d'erreurs de syntaxe au sein du programme tout en créant une représentation structurelle du programme qui permet de vérifier la sémantique de celui-ci. Cette structure se présente généralement, et dans notre cas, sous forme d'un arbre de syntaxe abstrait[1].

Par définition, si un programme a compilé, il est considéré comme correct dans le sens où il ne contient pas d'erreurs de syntaxe ou de sémantique. De ce fait, il nous est en réalité peu utile de vérifier la présence d'erreurs de syntaxe

et de sémantique. Cependant, nous avons jugé être une bonne chose de créer un couple *lexer/parser* pour, dans un premier temps, pouvoir vérifier si des programmes x86 codés manuellement ne contiennent pas d'erreurs (l'outil accepte donc des programmes binaires exécutables et des programmes en code assembleur x86) mais également car la représentation structurelle sous forme d'arbre de syntaxe abstrait nous sera indispensable pour créer nos clauses CLP. De plus, comme nous nous limitons à un sous-ensemble des programmes exécutables X86, cette association *lexer/parser* est particulièrement utile pour représenter ces limites (les instructions et conventions auxquelles nous nous limitons) tout en permettant d'étendre ce champs d'application sans trop de difficulté par la suite.

Dans notre cas, notre lexer analyse donc le code assembleur x86 épuré à l'étape précédente et renvoie des *tokens* afin d'indiquer si la suite de caractère analysée est un opérateur arithmétique, un opérande mémoire, un registre ou autre. Lorsque notre *parser* reçoit ces *tokens*, il vérifie donc la syntaxe des instructions afin d'éviter une erreur potentielle dans nos traductions en clauses CLP (dans le cas d'un programme x86 codé manuellement ou qui inclut une instruction que nous ne traitons pas encore).

Suite à cette vérification, notre *parser* crée également un arbre de syntaxe abstrait qui est une structure permettant de représenter le programme x86 reçu en entrée sous forme d'arbre dans le but de facilité la traduction en clauses CLP. Afin de créer notre arbre de syntaxe abstrait, il nous a donc fallut créer une grammaire conceptuelle et pour ce faire, nous avons décidé de définir une grammaire LL(1) correspondant aux programmes x86 que nous prenons en compte. Pour plus d'informations sur ces concepts utilisés par les compilateurs, [1] et [18] fournissent toutes les informations nécessaires à ce sujet.

⟨*program*⟩ : := ⟨*functions*⟩

⟨*blocks*⟩ : := ⟨*function*⟩ ⟨*functions*⟩ | ⟨*empty*⟩

⟨*function*⟩ : := LABEL ‘:’ ⟨*instrs*⟩

⟨*instrs*⟩ : := ⟨*instr*⟩ ⟨*instrs*⟩ | ⟨*empty*⟩

⟨*instr*⟩ : := OPERATOR terms

⟨*terms*⟩ : := ⟨*term*⟩ ⟨*terms*⟩ | ⟨*empty*⟩

⟨*term*⟩ : := REGISTER
| NUMBER
| DS : NUMBER
| ⟨*address*⟩

⟨*address*⟩ : := ‘[’REGISTER‘]’ | ‘[’NUMBER‘]’

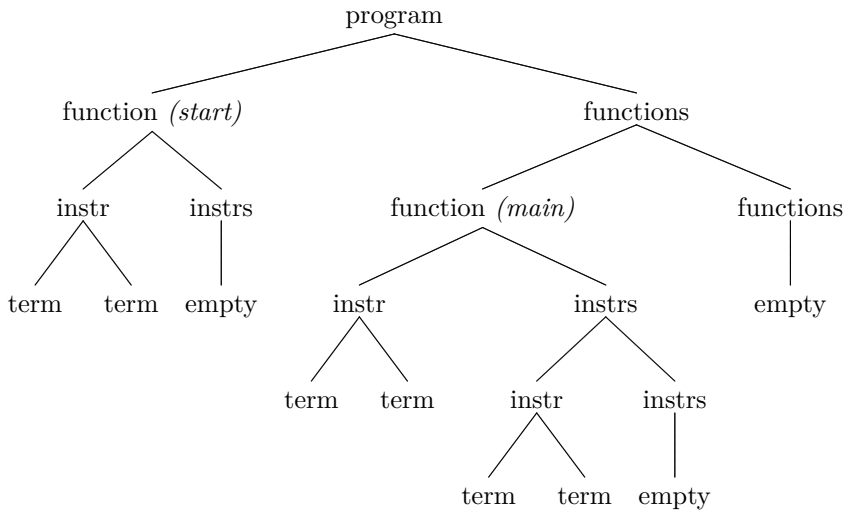
5.1.3 Arbre de syntaxe abstrait et traduction

Notre arbre syntaxique abstrait nous permet de vérifier la sémantique du programme x86 reçu en entrée. Une fois celle-ci vérifiée, l'arbre se crée au fur et à mesure de la lecture du code assembleur x86 en fonction de la grammaire

précédente de sorte à ce que chaque dernier nœud d'une branche soit un opérateur et que les fils de ce nœud (les feuilles) soient ses opérandes.

Ainsi, pour le programme x86 suivant, voici l'arbre de syntaxe abstrait que nous obtiendrions.

```
start :
   add eax, 3
main :
   mov ebx, eax
   sub ebx, 1
```



Une fois notre arbre syntaxique abstrait créé, nous avons pour chaque noeud créé une méthode permettant de le traduire en clauses CLP et ce, en fonction de l'approche que nous vous avons présentée jusqu'ici. De ce fait, notre traduction en clauses CLP se fait de manière récursive en appelant la méthode de traduction de la racine de l'arbre.

Nous obtenons finalement en sortie un fichier texte contenant une série de clauses CLP représentant chaque instruction x86 contenue dans le fichier binaire exécutable reçu en entrée.

5.2 Exemple

Cette section a pour but de conclure ce chapitre par l'illustration d'un exemple d'utilisation de l'outil Hornix afin d'avoir un aperçu d'une traduction complète d'un programme exécutable x86.

Pour ce faire, et également afin de donner un aperçu d'une potentielle utilisation du résultat de notre traduction, nous allons créer deux programmes C calculant l'exposant d'un nombre de manière différente. La première version calculera l'exposant par l'intermédiaire d'une boucle et la seconde version le

calculera de façon récursive.

Première version :

```
int exposant(int base, int exp) {
    int res = 1;
    for (int i=1; i<=exp; i++) {
        res = res * base;
    }
    return res;
}

int main() {
    int a = 3;
    int b = 5;
    return exposant(a,b);
}
```

```
080483db <exposant>:
 80483db: 55                    push   ebp
 80483dc: 89 e5                 mov    ebp,esp
 80483de: 83 ec 10              sub    esp,0x10
 80483e1: c7 45 f8 01 00 00 00  mov    DWORD PTR [ebp-0x8],0x1
 80483e8: c7 45 fc 01 00 00 00  mov    DWORD PTR [ebp-0x4],0x1
 80483ef: eb 0e                 jmp    80483ff <exposant+0x24>
 80483f1: 8b 45 f8              mov    eax,DWORD PTR [ebp-0x8]
 80483f4: 0f af 45 08           imul   eax,DWORD PTR [ebp+0x8]
 80483f8: 89 45 f8              mov    DWORD PTR [ebp-0x8],eax
 80483fb: 83 45 fc 01           add    DWORD PTR [ebp-0x4],0x1
 80483ff: 8b 45 fc              mov    eax,DWORD PTR [ebp-0x4]
 8048402: 3b 45 0c              cmp    eax,DWORD PTR [ebp+0xc]
 8048405: 7e ea                 jle    80483f1 <exposant+0x16>
 8048407: 8b 45 f8              mov    eax,DWORD PTR [ebp-0x8]
 804840a: c9                    leave
 804840b: c3                    ret

0804840c <main>:
 804840c: 55                    push   ebp
 804840d: 89 e5                 mov    ebp,esp
 804840f: 83 ec 10              sub    esp,0x10
 8048412: c7 45 f8 03 00 00 00  mov    DWORD PTR [ebp-0x8],0x3
 8048419: c7 45 fc 05 00 00 00  mov    DWORD PTR [ebp-0x4],0x5
 8048420: ff 75 fc              push   DWORD PTR [ebp-0x4]
 8048423: ff 75 f8              push   DWORD PTR [ebp-0x8]
 8048426: e8 b0 ff ff ff        call   80483db <exposant>
 804842b: 83 c4 08              add    esp,0x8
 804842e: c9                    leave
 804842f: c3                    ret
```

Deuxième version :

```
int exposant(int base, int exp) {
    if (exp <= 0) return 1;

    return base * exposant(base,exp-1);
}

int main() {
    int a = 3;
    int b = 5;
    return exposant(a,b);
}
```

```
080483db <exposant>:
 80483db: 55                    push   ebp
 80483dc: 89 e5                 mov    ebp,esp
 80483de: 83 ec 08              sub    esp,0x8
 80483e1: 83 7d 0c 00           cmp    DWORD PTR [ebp+0xc],0x0
 80483e5: 7f 07                 jg     80483ee <exposant+0x13>
 80483e7: b8 01 00 00 00        mov    eax,0x1
 80483ec: eb 19                 jmp    8048407 <exposant+0x2c>
 80483ee: 8b 45 0c              mov    eax,DWORD PTR [ebp+0xc]
 80483f1: 83 e8 01              sub    eax,0x1
 80483f4: 83 ec 08              sub    esp,0x8
 80483f7: 50                    push   eax
 80483f8: ff 75 08              push   DWORD PTR [ebp+0x8]
 80483fb: e8 db ff ff ff        call   80483db <exposant>
 8048400: 83 c4 10              add    esp,0x10
 8048403: 0f af 45 08           imul   eax,DWORD PTR [ebp+0x8]
 8048407: c9                    leave
 8048408: c3                    ret

08048409 <main>:
 8048409: 8d 4c 24 04           lea    ecx,[esp+0x4]
 804840d: 83 e4 f0              and    esp,0xfffffff0
 8048410: ff 71 fc              push   DWORD PTR [ecx-0x4]
 8048413: 55                    push   ebp
 8048414: 89 e5                 mov    ebp,esp
 8048416: 51                    push   ecx
 8048417: 83 ec 14              sub    esp,0x14
 804841a: c7 45 f0 03 00 00 00  mov    DWORD PTR [ebp-0x10],0x3
 8048421: c7 45 f4 05 00 00 00  mov    DWORD PTR [ebp-0xc],0x5
 8048428: 83 ec 08              sub    esp,0x8
 804842b: ff 75 f4              push   DWORD PTR [ebp-0xc]
 804842e: ff 75 f0              push   DWORD PTR [ebp-0x10]
 8048431: e8 a5 ff ff ff        call   80483db <exposant>
 8048436: 83 c4 10              add    esp,0x10
```

```
8048439: 8b 4d fc             mov    ecx,DWORD PTR [ebp-0x4]
804843c: c9                   leave
804843d: 8d 61 fc             lea    esp,[ecx-0x4]
8048440: c3                   ret
8048441: 66 90                xchg   ax,ax
8048443: 66 90                xchg   ax,ax
8048445: 66 90                xchg   ax,ax
8048447: 66 90                xchg   ax,ax
8048449: 66 90                xchg   ax,ax
804844b: 66 90                xchg   ax,ax
804844d: 66 90                xchg   ax,ax
804844f: 90                   nop
```

Voici le code CLP obtenu pour la première version.

```
% Program variables :

% Program functions :
% exposant = l4
% main = l3

p_start(R)
:- l3(EAX,EBX,ECX,EDX,EBP,ESP,R).


l4(EAX,EBX,ECX,EDX,EBP,ESP,R,S0,S1)
:- hornix1(EAX,EBX,ECX,EDX,EBP,ESP,R,S0,S1).

hornix1(EAX,EBX,ECX,EDX,EBP,ESP,R,S0,S1)
:- hornix2(EAX,EBX,ECX,EDX,EBP,ESP,R,S0,S1,RET).

hornix2(EAX,EBX,ECX,EDX,EBP,ESP,R,S0,S1,S2)
:- hornix3(EAX,EBX,ECX,EDX,EBP,ESP,R,S0,S1,S2,EBP).

hornix3(EAX,EBX,ECX,EDX,EBP,ESP,R,S0,S1,S2,S3)
:- hornix4(EAX,EBX,ECX,EDX,ESP,ESP,R,S0,S1,S2,S3).

hornix4(EAX,EBX,ECX,EDX,EBP,ESP,R,S0,S1,S2,S3)
:- hornix5(EAX,EBX,ECX,EDX,EBP,ESP,R,S0,S1,S2,S3,NULL,NULL,NULL,NULL).

hornix5(EAX,EBX,ECX,EDX,EBP,ESP,R,S0,S1,S2,S3,S4,S5,S6,S7)
:- hornix6(EAX,EBX,ECX,EDX,EBP,ESP,R,S0,S1,S2,S3,S4,1,S6,S7).

hornix6(EAX,EBX,ECX,EDX,EBP,ESP,R,S0,S1,S2,S3,S4,S5,S6,S7)
:- hornix7(EAX,EBX,ECX,EDX,EBP,ESP,R,S0,S1,S2,S3,1,S5,S6,S7).

hornix7(EAX,EBX,ECX,EDX,EBP,ESP,R,S0,S1,S2,S3,S4,S5,S6,S7)
:- l1(EAX,EBX,ECX,EDX,EBP,ESP,R,S0,S1,S2,S3,S4,S5,S6,S7).
```

```
l2(EAX,EBX,ECX,EDX,EBP,ESP,R,S0,S1,S2,S3,S4,S5,S6,S7)
:- hornix8(EAX,EBX,ECX,EDX,EBP,ESP,R,S0,S1,S2,S3,S4,S5,S6,S7).

hornix8(EAX,EBX,ECX,EDX,EBP,ESP,R,S0,S1,S2,S3,S4,S5,S6,S7)
:- hornix9(S5,EBX,ECX,EDX,EBP,ESP,R,S0,S1,S2,S3,S4,S5,S6,S7).

hornix9(EAX,EBX,ECX,EDX,EBP,ESP,R,S0,S1,S2,S3,S4,S5,S6,S7)
:- hornix10((EAX) * (S1),EBX,ECX,EDX,EBP,ESP,R,S0,S1,S2,S3,S4,S5,S6,S7).

hornix10(EAX,EBX,ECX,EDX,EBP,ESP,R,S0,S1,S2,S3,S4,S5,S6,S7)
:- hornix11(EAX,EBX,ECX,EDX,EBP,ESP,R,S0,S1,S2,S3,S4,EAX,S6,S7).

hornix11(EAX,EBX,ECX,EDX,EBP,ESP,R,S0,S1,S2,S3,S4,S5,S6,S7)
:- hornix12(EAX,EBX,ECX,EDX,EBP,ESP,R,S0,S1,S2,S3,(S4) + (1),S5,S6,S7).


l1(EAX,EBX,ECX,EDX,EBP,ESP,R,S0,S1,S2,S3,S4,S5,S6,S7)
:- hornix12(EAX,EBX,ECX,EDX,EBP,ESP,R,S0,S1,S2,S3,S4,S5,S6,S7).

hornix12(EAX,EBX,ECX,EDX,EBP,ESP,R,S0,S1,S2,S3,S4,S5,S6,S7)
:- hornix13(S4,EBX,ECX,EDX,EBP,ESP,R,S0,S1,S2,S3,S4,S5,S6,S7).

hornix13(EAX,EBX,ECX,EDX,EBP,ESP,R,S0,S1,S2,S3,S4,S5,S6,S7)
:- {EAX =< S0}, l2(EAX,EBX,ECX,EDX,EBP,ESP,R,S0,S1,S2,S3,S4,S5,S6,S7).

hornix13(EAX,EBX,ECX,EDX,EBP,ESP,R,S0,S1,S2,S3,S4,S5,S6,S7)
:- {EAX > S0}, hornix14(EAX,EBX,ECX,EDX,EBP,ESP,R,S0,S1,S2,S3,S4,S5,S6,S7).

hornix14(EAX,EBX,ECX,EDX,EBP,ESP,R,S0,S1,S2,S3,S4,S5,S6,S7)
:- hornix15(S5,EBX,ECX,EDX,EBP,ESP,R,S0,S1,S2,S3,S4,S5,S6,S7).

hornix15(EAX,EBX,ECX,EDX,EBP,ESP,R,S0,S1,S2,S3,S4,S5,S6,S7)
:- hornix16(EAX,EBX,ECX,EDX,S7,ESP,R,S0,S1,S2,S3,S4,S5,S6).

hornix16(EAX,EBX,ECX,EDX,EBP,ESP,R,S0,S1,S2,S3,S4,S5,S6)
:- {R = EAX}.


l3(EAX,EBX,ECX,EDX,EBP,ESP,R)
:- hornix17(EAX,EBX,ECX,EDX,EBP,ESP,R).

hornix17(EAX,EBX,ECX,EDX,EBP,ESP,R)
:- hornix18(EAX,EBX,ECX,EDX,EBP,ESP,R,EBP).

hornix18(EAX,EBX,ECX,EDX,EBP,ESP,R,S0)
:- hornix19(EAX,EBX,ECX,EDX,ESP,ESP,R,S0).

hornix19(EAX,EBX,ECX,EDX,EBP,ESP,R,S0)
```

```
:- hornix20(EAX,EBX,ECX,EDX,EBP,ESP,R,S0,NULL,NULL,NULL,NULL).

hornix20(EAX,EBX,ECX,EDX,EBP,ESP,R,S0,S1,S2,S3,S4)
:- hornix21(EAX,EBX,ECX,EDX,EBP,ESP,R,S0,S1,3,S3,S4).

hornix21(EAX,EBX,ECX,EDX,EBP,ESP,R,S0,S1,S2,S3,S4)
:- hornix22(EAX,EBX,ECX,EDX,EBP,ESP,R,S0,5,S2,S3,S4).

hornix22(EAX,EBX,ECX,EDX,EBP,ESP,R,S0,S1,S2,S3,S4)
:- hornix23(EAX,EBX,ECX,EDX,EBP,ESP,R,S0,S1,S2,S3,S4,S1).

hornix23(EAX,EBX,ECX,EDX,EBP,ESP,R,S0,S1,S2,S3,S4,S5)
:- hornix24(EAX,EBX,ECX,EDX,EBP,ESP,R,S0,S1,S2,S3,S4,S5,S2).

hornix24(EAX,EBX,ECX,EDX,EBP,ESP,R,S0,S1,S2,S3,S4,S5,S6)
:- l4(EAX,EBX,ECX,EDX,EBP,ESP,TMP,S5,S6),
   hornix25(TMP,EBX,ECX,EDX,EBP,ESP,R,S0,S1,S2,S3,S4).

hornix25(EAX,EBX,ECX,EDX,EBP,ESP,R,S0,S1,S2,S3,S4)
:- hornix26(EAX,EBX,ECX,EDX,S4,ESP,R,S0,S1,S2,S3).

hornix26(EAX,EBX,ECX,EDX,EBP,ESP,R,S0,S1,S2,S3)
:- {R = EAX}.
```

Voici le code CLP obtenu pour la seconde version.

```
% Program variables :

% Program functions :
% exposant = l3
% main = l4

p_start(R)
:- l4(EAX,EBX,ECX,EDX,EBP,ESP,R).


l3(EAX,EBX,ECX,EDX,EBP,ESP,R,S0,S1,S2,S3)
:- hornix1(EAX,EBX,ECX,EDX,EBP,ESP,R,S0,S1,S2,S3).

hornix1(EAX,EBX,ECX,EDX,EBP,ESP,R,S0,S1,S2,S3)
:- hornix2(EAX,EBX,ECX,EDX,EBP,ESP,R,S0,S1,S2,S3,RET).

hornix2(EAX,EBX,ECX,EDX,EBP,ESP,R,S0,S1,S2,S3,S4)
:- hornix3(EAX,EBX,ECX,EDX,EBP,ESP,R,S0,S1,S2,S3,S4,EBP).

hornix3(EAX,EBX,ECX,EDX,EBP,ESP,R,S0,S1,S2,S3,S4,S5)
:- hornix4(EAX,EBX,ECX,EDX,ESP,ESP,R,S0,S1,S2,S3,S4,S5).
```

```
hornix4(EAX,EBX,ECX,EDX,EBP,ESP,R,S0,S1,S2,S3,S4,S5)
:- hornix5(EAX,EBX,ECX,EDX,EBP,ESP,R,S0,S1,S2,S3,S4,S5,NULL,NULL).

hornix5(EAX,EBX,ECX,EDX,EBP,ESP,R,S0,S1,S2,S3,S4,S5,S6,S7)
:- {S2 > 0}, l1(EAX,EBX,ECX,EDX,EBP,ESP,R,S0,S1,S2,S3,S4,S5,S6,S7).

hornix5(EAX,EBX,ECX,EDX,EBP,ESP,R,S0,S1,S2,S3,S4,S5,S6,S7)
:- {S2 =< 0}, hornix6(EAX,EBX,ECX,EDX,EBP,ESP,R,S0,S1,S2,S3,S4,S5,S6,S7).

hornix6(EAX,EBX,ECX,EDX,EBP,ESP,R,S0,S1,S2,S3,S4,S5,S6,S7)
:- hornix7(1,EBX,ECX,EDX,EBP,ESP,R,S0,S1,S2,S3,S4,S5,S6,S7).

hornix7(EAX,EBX,ECX,EDX,EBP,ESP,R,S0,S1,S2,S3,S4,S5,S6,S7)
:- l2(EAX,EBX,ECX,EDX,EBP,ESP,R,S0,S1,S2,S3,S4,S5,S6,S7).


l1(EAX,EBX,ECX,EDX,EBP,ESP,R,S0,S1,S2,S3,S4,S5,S6,S7)
:- hornix8(EAX,EBX,ECX,EDX,EBP,ESP,R,S0,S1,S2,S3,S4,S5,S6,S7).

hornix8(EAX,EBX,ECX,EDX,EBP,ESP,R,S0,S1,S2,S3,S4,S5,S6,S7)
:- hornix9(S2,EBX,ECX,EDX,EBP,ESP,R,S0,S1,S2,S3,S4,S5,S6,S7).

hornix9(EAX,EBX,ECX,EDX,EBP,ESP,R,S0,S1,S2,S3,S4,S5,S6,S7)
:- hornix10((EAX) - (1),EBX,ECX,EDX,EBP,ESP,R,S0,S1,S2,S3,S4,S5,S6,S7).

hornix10(EAX,EBX,ECX,EDX,EBP,ESP,R,S0,S1,S2,S3,S4,S5,S6,S7)
:- hornix11(EAX,EBX,ECX,EDX,EBP,ESP,R,S0,S1,S2,S3,S4,S5,S6,S7,NULL,NULL).

hornix11(EAX,EBX,ECX,EDX,EBP,ESP,R,S0,S1,S2,S3,S4,S5,S6,S7,S8,S9)
:- hornix12(EAX,EBX,ECX,EDX,EBP,ESP,R,S0,S1,S2,S3,S4,S5,S6,S7,S8,S9,EAX).

hornix12(EAX,EBX,ECX,EDX,EBP,ESP,R,S0,S1,S2,S3,S4,S5,S6,S7,S8,S9,S10)
:- hornix13(EAX,EBX,ECX,EDX,EBP,ESP,R,S0,S1,S2,S3,S4,S5,S6,S7,S8,S9,S10,S3).

hornix13(EAX,EBX,ECX,EDX,EBP,ESP,R,S0,S1,S2,S3,S4,S5,S6,S7,S8,S9,S10,S11)
:- l3(EAX,EBX,ECX,EDX,EBP,ESP,TMP,S8,S9,S10,S11),
   hornix14(TMP,EBX,ECX,EDX,EBP,ESP,R,S0,S1,S2,S3,S4,S5,S6,S7).

hornix14(EAX,EBX,ECX,EDX,EBP,ESP,R,S0,S1,S2,S3,S4,S5,S6,S7)
:- hornix15((EAX) * (S3),EBX,ECX,EDX,EBP,ESP,R,S0,S1,S2,S3,S4,S5,S6,S7).


l2(EAX,EBX,ECX,EDX,EBP,ESP,R,S0,S1,S2,S3,S4,S5,S6,S7)
:- hornix15(EAX,EBX,ECX,EDX,EBP,ESP,R,S0,S1,S2,S3,S4,S5,S6,S7).

hornix15(EAX,EBX,ECX,EDX,EBP,ESP,R,S0,S1,S2,S3,S4,S5,S6,S7)
:- hornix16(EAX,EBX,ECX,EDX,S7,ESP,R,S0,S1,S2,S3,S4,S5,S6).

hornix16(EAX,EBX,ECX,EDX,EBP,ESP,R,S0,S1,S2,S3,S4,S5,S6)
:- {R = EAX}.
```

```
l4(EAX,EBX,ECX,EDX,EBP,ESP,R)
:- hornix17(EAX,EBX,ECX,EDX,EBP,ESP,R).

hornix17(EAX,EBX,ECX,EDX,EBP,ESP,R)
:- hornix18(EAX,EBX,ECX,EDX,EBP,ESP,R,EBP).

hornix18(EAX,EBX,ECX,EDX,EBP,ESP,R,S0)
:- hornix19(EAX,EBX,ECX,EDX,ESP,ESP,R,S0).

hornix19(EAX,EBX,ECX,EDX,EBP,ESP,R,S0)
:- hornix20(EAX,EBX,ECX,EDX,EBP,ESP,R,S0,ECX).

hornix20(EAX,EBX,ECX,EDX,EBP,ESP,R,S0,S1)
:- hornix21(EAX,EBX,ECX,EDX,EBP,ESP,R,S0,S1,NULL,NULL,NULL,NULL,NULL).

hornix21(EAX,EBX,ECX,EDX,EBP,ESP,R,S0,S1,S2,S3,S4,S5,S6)
:- hornix22(EAX,EBX,ECX,EDX,EBP,ESP,R,S0,S1,S2,S3,3,S5,S6).

hornix22(EAX,EBX,ECX,EDX,EBP,ESP,R,S0,S1,S2,S3,S4,S5,S6)
:- hornix23(EAX,EBX,ECX,EDX,EBP,ESP,R,S0,S1,S2,5,S4,S5,S6).

hornix23(EAX,EBX,ECX,EDX,EBP,ESP,R,S0,S1,S2,S3,S4,S5,S6)
:- hornix24(EAX,EBX,ECX,EDX,EBP,ESP,R,S0,S1,S2,S3,S4,S5,S6,NULL,NULL).

hornix24(EAX,EBX,ECX,EDX,EBP,ESP,R,S0,S1,S2,S3,S4,S5,S6,S7,S8)
:- hornix25(EAX,EBX,ECX,EDX,EBP,ESP,R,S0,S1,S2,S3,S4,S5,S6,S7,S8,S3).

hornix25(EAX,EBX,ECX,EDX,EBP,ESP,R,S0,S1,S2,S3,S4,S5,S6,S7,S8,S9)
:- hornix26(EAX,EBX,ECX,EDX,EBP,ESP,R,S0,S1,S2,S3,S4,S5,S6,S7,S8,S9,S4).

hornix26(EAX,EBX,ECX,EDX,EBP,ESP,R,S0,S1,S2,S3,S4,S5,S6,S7,S8,S9,S10)
:- l3(EAX,EBX,ECX,EDX,EBP,ESP,TMP,S7,S8,S9,S10),
   hornix27(TMP,EBX,ECX,EDX,EBP,ESP,R,S0,S1,S2,S3,S4,S5,S6).

hornix27(EAX,EBX,ECX,EDX,EBP,ESP,R,S0,S1,S2,S3,S4,S5,S6)
:- hornix28(EAX,EBX,S1,EDX,EBP,ESP,R,S0,S1,S2,S3,S4,S5,S6).

hornix28(EAX,EBX,ECX,EDX,EBP,ESP,R,S0,S1,S2,S3,S4,S5,S6)
:- hornix29(EAX,EBX,ECX,EDX,S6,ESP,R,S0,S1,S2,S3,S4,S5).

hornix29(EAX,EBX,ECX,EDX,EBP,ESP,R,S0,S1,S2,S3,S4,S5)
:- hornix30(EAX,EBX,ECX,EDX,TMP,ESP,R,S0,S1,S2,S3,S4,S5).

hornix30(EAX,EBX,ECX,EDX,EBP,ESP,R,S0,S1,S2,S3,S4,S5)
:- {R = EAX}.
```

Pour illustrer une des utilités de nos résultats, l'application de l'outil [26] sur ces clauses de Horn révélera que les deux programmes renvoient bien le même résultat mais que ceux-ci utilisent des algorithmes différents.

Chapitre 6

Améliorations et travaux futurs

Comme déjà énoncé à plusieurs reprises, durant nos recherches, nous nous sommes limités à un sous-ensemble de programmes exécutables x86 en fonction des conventions et des limites que nous respections. Nous donnons dans ce chapitre quelques fonctionnalités du code assembleur x86 qu'il serait intéressant d'implémenter afin d'agrandir le champs d'application de notre étude.

6.1 Les librairies externes

Lorsque des programmes sont implémentés en langage de haut niveau, il est presque inévitable de faire appel à des librairies externes afin de réaliser une tâche bien précise, c'est tout logiquement le cas pour des programmes implémentés en C.

Durant nos recherches, nous n'avons pas pris en compte ces appels à des librairies externes car, de nos jours, les compilateurs utilisent une méthode dite de liaison dynamique afin d'inclure les librairies utilisées dans le programme à compiler et cette méthode est particulièrement difficile à représenter et nécessite des connaissances qui dépassent notre domaine de recherche actuel.

Prenons l'exemple du programma C suivant.

```
#include <stdio.h>
#include <string.h>

int main() {
        int a = 5;

        printf("%d\n",a);

        return 0;
}
```

Le simple fait d'appeler la fonction *printf* nécessite d'inclure la librairie *stdio.h*.

Aux débuts des compilateurs, ceux-ci utilisaient une méthode de liaison statique pour inclure les librairies utilisées par un programme dans la compilation. Cela signifie que, lors de la compilation d'un programme, les librairies requises par ce dernier étaient incluses et compilées au sein du même fichier exécutable. Cela avait comme avantage que le code assembleur x86 des librairies étaient intégralement inclus dans le fichier exécutable, c'est-à-dire dans la section *.text* (les appels sont donc déjà résolus et de ce fait, immédiats) mais cela avait également comme principale désavantage de rendre les fichiers exécutables extrêmement lourds.

C'est pourquoi, de nos jours, les compilateurs utilisent une méthode de liaison dynamique. Cela signifie qu'ils n'incluent plus les librairies requises dans l'exécutable final, ils n'incluent maintenant plus que le nom du symbole (la fonction), ainsi que le nom de la librairie utilisée. De ce fait, c'est à l'exécution du programme que l'éditeur de liens dynamique (*dynamic linker*) effectue la résolution des symboles. Voici ci-dessous le code assembleur x86 correspondant au programme ci-dessus, compilé des deux façons possibles. A titre de comparaison, l'exécutable compilé dynamiquement pèse 7,4ko et l'exécutable compilé statiquement pèse 725,3ko.

Compilation statique :

```
0804ebf0 <_IO_printf>:
 ...


0804887c <main>:
 ...
 8048886: 55                    push   ebp
 8048887: 89 e5                 mov    ebp,esp
 8048889: 51                    push   ecx
 804888a: 83 ec 14              sub    esp,0x14
 804888d: c7 45 f4 05 00 00 00  mov    DWORD PTR [ebp-0xc],0x5
 8048894: 83 ec 08              sub    esp,0x8
 8048897: ff 75 f4              push   DWORD PTR [ebp-0xc]
 804889a: 68 c8 ac 0b 08        push   0x80bacc8
 804889f: e8 4c 63 00 00        call   804ebf0 <_IO_printf>
 80488a4: 83 c4 10              add    esp,0x10
 80488a7: b8 00 00 00 00        mov    eax,0x0
 80488ac: 8b 4d fc              mov    ecx,DWORD PTR [ebp-0x4]
 80488af: c9                    leave
 80488b0: 8d 61 fc              lea    esp,[ecx-0x4]
 80488b3: c3                    ret
 ...
```

Compilation dynamique :

```
0804840b <main>:
 ...
 8048415: 55                    push   ebp
 8048416: 89 e5                 mov    ebp,esp
 8048418: 51                    push   ecx
 8048419: 83 ec 14              sub    esp,0x14
 804841c: c7 45 f4 05 00 00 00  mov    DWORD PTR [ebp-0xc],0x5
 8048423: 83 ec 08              sub    esp,0x8
 8048426: ff 75 f4              push   DWORD PTR [ebp-0xc]
 8048429: 68 d0 84 04 08        push   0x80484d0
 804842e: e8 ad fe ff ff        call   80482e0 <printf@plt>
 8048433: 83 c4 10              add    esp,0x10
 8048436: b8 00 00 00 00        mov    eax,0x0
 804843b: 8b 4d fc              mov    ecx,DWORD PTR [ebp-0x4]
 804843e: c9                    leave
 804843f: 8d 61 fc              lea    esp,[ecx-0x4]
 8048442: c3                    ret
 ...
```

Nous pouvons constater que dans la compilation statique, l'appel à la fonction *printf* correspond bien à un appel de fonction tel que nous les avons traités. Cependant, concernant la liaison dynamique, le label *80482e0* ne se trouve pas dans la section *.text*, contrairement à la liaison statique mais dans une section nommé *.plt*.

La section *.plt* forme une paire avec la section *.got*, leur rôle est consacré à aider l'éditeur de liens dynamique à résoudre les différents appels. Sans rentrer dans les détails de la liaison dynamique et du fonctionnement de l'éditeur de liens car ce n'est pas le but de ce chapitre, ces deux sections contiennent deux tables qui sont la *Global Offset Table* et la *Procedure Linkage Table* et dont le rôle est d'indiquer à l'éditeur de liens dynamique quel symbole (fonction) il doit fournir afin de résoudre l'appel[4].

Le principal problème auquel nous sommes confrontés, en hormis le fait que la communication entre ces deux tables manipule des pointeurs que nous ne prenons pas en compte dans notre recherche, est que l'éditeur de lien effectue la résolution des appels au moment de l'exécution du programme. Cela signifie que, notre outil analysant des fichiers binaires sans les exécuter, il nous est donc impossible de tirer profit de la résolution de lien effectuée par l'éditeur de liens dynamiques.

Notre méthode ne permet donc pas, à l'heure actuelle, de traiter des fichiers binaires exécutables x86 nécessitant des librairies externes et ayant été compilés dynamiquement. Il serait donc intéressant d'approfondir les recherches sur cet aspect-ci afin d'agrandir l'ensemble de programmes x86 sur lequel s'applique notre méthode de traduction.

6.2 Les opérations sur les bits

Le code assembleur x86 étant un langage bas niveau et travaillant avec des registres, celui-ci contient un nombre non-négligeable d'opérations binaires. En effet, il permet à la fois de travailler avec des nombres décimaux et hexadécimaux mais il permet également de manipuler les bits des registres et comporte également des opérateurs binaires comme le "ET" logique, le "OU" logique,...

Nous avons brièvement abordé le sujet lorsque nous appliquions notre approche de traduction à la l'opération arithmétique de division dans la section 4.1.1. La programmation logique par contraintes reste un paradigme de programmation et n'est pas encore un langage de programmation concret et de ce fait, notre outil Hornix devant renvoyer un résultat concret, nous avons fait le choix d'utiliser une représentation et un principe de fonctionnement proche de Prolog afin de rendre ce paradigme le plus réaliste possible.

De ce fait, pour traduire des opérations binaires comme *shift-left*, la conjonction ou la disjonction de deux registres, nous pourrions supposer et poser l'hypothèse que la programmation logique par contraintes accepte et gère les opérations binaires aussi bien que les opérations décimales mais afin de rester fidèle à notre choix de rendre ce paradigme le plus concret possible, il serait intéressant de trouver un moyen plus concret de représenter ces opérations binaires.
Cela dit, cette première solution est tout à fait envisageable car certaines librairies Prolog permettent de gérer en partie des opérations logiques comme la conjonction et la disjonction de deux nombres[23].

Une autre solution est celle que nous avons utilisée pour représenter la division. Cette solution consiste à supposer que Prolog et la programmation logique par contrainte ne gère pas les nombres et opérations binaires et de ce fait, qu'il est donc nécessaire de trouver un moyen de traduire une opération binaire en une suite d'opérations arithmétiques décimales. Pour certains instructions comme les déplacements de bits (*shift-left, shift-right,...*, cela est facilement faisable mais pour d'autres instructions, cela se complique beaucoup et nous nous confrontons à certains moments à des problèmes techniques. Un exemple de ces problèmes est que si l'implémentation de l'environnement logique dans lequel nous travaillons (SWI-Prolog[24] pour Prolog, par exemple) est en 32 bits, l'opération 2^{32} utilisée pour traduire la division en clauses CLP devient problématique à réaliser.

Enfin, la figure 6.1 illustre les fonctions arithmétiques décimales qui traduisent les opérations logiques de base. Cela illustre également la complexité de cette solution.

6.3 les adressages directs

Nous avons vu que le code assembleur x86 faisait souvent usage des adresses mémoires dans ses instructions, que ce soit pour effectuer des instructions de branchement ou bien aller chercher une donnée.

$$\mathrm{NOT}\, x = \sum_{n=0}^{\lfloor \log_2(x) \rfloor} 2^n \left[\left(\left\lfloor \frac{x}{2^n} \right\rfloor \bmod 2 + 1 \right) \bmod 2 \right] = 2^{\lfloor \log_2(x) \rfloor + 1} - 1 - x$$

$$x\ \mathrm{AND}\ y = \sum_{n=0}^{\lfloor \log_2(x) \rfloor} 2^n \left(\left\lfloor \frac{x}{2^n} \right\rfloor \bmod 2 \right) \left(\left\lfloor \frac{y}{2^n} \right\rfloor \bmod 2 \right)$$

$$x\ \mathrm{OR}\ y = \sum_{n=0}^{\lfloor \log_2(x) \rfloor} 2^n \left[\left[\left(\left\lfloor \frac{x}{2^n} \right\rfloor \bmod 2 \right) + \left(\left\lfloor \frac{y}{2^n} \right\rfloor \bmod 2 \right) + \left(\left\lfloor \frac{x}{2^n} \right\rfloor \bmod 2 \right) \left(\left\lfloor \frac{y}{2^n} \right\rfloor \bmod 2 \right) \right] \bmod 2 \right]$$

$$x\ \mathrm{XOR}\ y = \sum_{n=0}^{\lfloor \log_2(x) \rfloor} 2^n \left[\left[\left(\left\lfloor \frac{x}{2^n} \right\rfloor \bmod 2 \right) + \left(\left\lfloor \frac{y}{2^n} \right\rfloor \bmod 2 \right) \right] \bmod 2 \right] = \sum_{n=0}^{\lfloor \log_2(x) \rfloor} 2^n \left[\left(\left\lfloor \frac{x}{2^n} \right\rfloor + \left\lfloor \frac{y}{2^n} \right\rfloor \right) \bmod 2 \right]$$

FIGURE 6.1 – Fonctions arithmétiques décimales traduisant certaines opérations[25]

Lorsqu'une adresse mémoire défini un élément ou un endroit précis du code assembleur que nous analysons, il est possible de le représenter en clauses CLP comme nous l'avons fait pour l'utilisation de variables globales et de constantes. Cependant, ces adresses mémoires ne font pas toujours partie du code assembleur que nous analysons et dans ce cas, il nous est alors beaucoup plus difficile de les traiter et de les traduire en clauses CLP. Une des instructions x86 manipulant des adresses que nous n'avons pas encore implémenté dans notre outil mais qu'il serait intéressant de représenter en clauses CLP est l'instruction *LEA*.

Cet opérateur nécessite deux opérandes et place dans son premier opérande l'adresse mémoire effective du second opérande. Ainsi, l'instruction *LEA EAX, [EBP-8]* place dans *EAX*, non pas l'élément *EBP-8* se trouvant sur la pile mais son adresse mémoire effective. Afin de mieux comprendre, nous pouvons assimiler ça à l'utilisation d'un pointeur.

Cette instruction est utilisée pour plusieurs raison dont une qui est le passage de tableau en argument d'une fonction appelée. En effet, lorsqu'un programme C initialise un tableau, le code assembleur le représente sur la pile de sorte que si le tableau comporte 5 éléments, ces 5 éléments seront empilés sur la pile l'un à la suite de l'autre. Ainsi, l'index qu'utilise le programme afin de manipuler un élément spécifique du tableau est équivalent à l'index soustrait à *EBP* pour retrouver un élément sur la pile en x86. Cette utilisation de tableau en C est donc bien prise en compte par notre méthode de traduction.

Cependant, lorsque la fonction *main* d'un programme C appelle une fonction nécessitant un tableau en paramètre, le code assembleur x86 ne va pas ré-empiler une seconde fois les éléments du tableau afin de les passer en paramètre comme nous l'avons vu. Le code assembleur va utiliser l'instruction *LEA* de manière à empiler l'adresse mémoire effective du premier élément du tableau qu'il a empilé afin de passer cette adresse effective comme paramètre à la fonction appelée. Ainsi, lorsque la fonction C appelée manipulera un élément précis du tableau via un index, cela ne se fera, non plus en soustrayant cet index à *EBP*, mais en le soustrayant à l'adresse effective fournie par la fonction appelante (ici, *main*) car les éléments du tableau ont été empilés l'un après l'autre.

Afin de rendre ces explications plus claires, voici un exemple de programme C et son désassemblage commenté.

```
int addArray23(int arr[]) {
        return arr[2]+arr[3];
}

int main() {
        int array[] = {5,6,8,2,6};
        return addArray23(array);
}
```

```
...
addArray23:
  push   ebp
  mov    ebp,esp
  mov    eax,DWORD PTR [ebp+0x8]  ;EAX = paramètre (adresse mémoire de l'élément 5)
  add    eax,0x8                  ;EAX = (adresse de 5) + 8
                                       = deux emplacements mémoire après l'adresse de 5
                                       = adresse mémoire de 8 (pointeur)

  mov    edx,DWORD PTR [eax]      ;EDX = élément pointé par l'adresse obtenu = 8
  mov    eax,DWORD PTR [ebp+0x8]  ;EAX = paramètre (adresse mémoire de l'élément 5)
  add    eax,0xc                  ;EAX = (adresse de 5) + 12
                                       = trois emplacements mémoire après l'adresse de 5
                                       = adresse mémoire de 2 (pointeur)

  mov    eax,DWORD PTR [eax]      ;EAX = élément pointé par l'adresse obtenu = 2
  add    eax,edx                  ;EAX = EAX + EDX = 2 + 8 = 10
  pop    ebp
  ret


...
main:
  ...
  mov    DWORD PTR [ebp-0x20],0x5  ;empilement de 5 à EBP-32
  mov    DWORD PTR [ebp-0x1c],0x6  ;empilement de 6 à EBP-28
  mov    DWORD PTR [ebp-0x18],0x8  ;empilement de 8 à EBP-24
  mov    DWORD PTR [ebp-0x14],0x2  ;empilement de 2 à EBP-20
  mov    DWORD PTR [ebp-0x10],0x6  ;empilement de 6 à EBP-16
  lea    eax,[ebp-0x20]            ;EAX = adresse mémoire de l'élément 5 (pointeur)
  push   eax                       ;empilement de EAX, passage de paramètre
  call   addArray23                ;appel de la fonction
  add    esp,0x4                   ;suppression de du paramètre précédemment empilé
  ...
```

Un passage de tableau en argument d'une fonction étant une pratique courante en programmation, il serait donc intéressant d'approfondir nos recherches sur ce type d'utilisation d'adresses mémoire.

Chapitre 7

Conclusion

En conclusion de ce mémoire, nous nous sommes donc intéressé à l'analyse des exécutables x86. Nous avons vu qu'il en existait de plusieurs types et afin d'obtenir des résultats non-négligeables, nous avons fait le choix de nous restreindre sur les exécutables au format ELF.

Le code assembleur x86, se devant d'être rétrocompatible, est devenu de nos jours extrêmement complexe à comprendre et à interpréter dans ses aspects les plus profonds. L'objectif de nos recherches étaient donc de rendre des analyses de programme plus facilement réalisable sur ce genre d'exécutable.

Pour ce faire, plutôt que d'essayer d'appliquer des analyses de programme précises sur un code assembleur x86, nous avons opté pour une solution qui permettrait de simplifier l'application générale d'analyses de programme. Cette solution réside en l'utilisation de la programmation logique par contrainte, pour laquelle les analyses de programme sont plus facilement réalisable. Plus concrètement, notre idée générale consiste à traduire un code assembleur x86 en un équivalent en programmation logique par contraintes.

Afin de concrétiser notre idée et de l'implémenter au sein d'un outil, nous avons développé une approche de traduction séquentielle où chaque instruction x86 serait représentée (traduite) par une ou plusieurs clauses CLP (*Constraint Logic Programming*. Nous avons donc présenté dans ce mémoire notre approche de traduction et comment celle-ci s'appliquait à un ensemble d'instructions x86.

Cependant, le code assembleur x86 étant un langage de bas niveau, il est très permissif et de ce fait, il est extrêmement compliqué de prendre en compte l'ensemble de tous les programmes exécutables x86. C'est pourquoi nous avons dû faire des choix et imposer des limites d'utilisation (respect de certaines conventions) quant aux programmes x86 que nous pouvons traiter avec notre méthode.

Notre méthode ne se veut pas être une méthode exhaustive mais se veut plutôt être une base solide et modulable afin de pouvoir, dans un futur proche, être capable de traduire un plus large ensemble de programmes x86. Pour l'instant, nous nous sommes donc restreints à un ensemble de programmes C simples, codés manuellement, et compilé en 32 bits.

Les résultats auxquels nous sommes arrivés sont intéressants car nous avons pu démontrer que notre méthode de traduction permettait de traduire un en-

semble d'instructions x86 relativement souvent utilisées par les compilateurs et car nous avons également été capables de représenter des concepts indispensables au bon fonctionnement du code assembleur x86 tels que la pile.

Enfin, de nombreuses améliorations restent donc encore à effectuer afin d'élargir l'ensemble de programmes x86 que nous prenons en compte mais nous estimons avoir conçu une méthode solide et modulable dans le sens où elle a été conçue et pensée dans le but de pouvoir incorporer ces améliorations sans nécessiter de lourds changements et modifications.

Bibliographie

[1] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers : Principles, Techniques, and Tools (2Nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.

[2] Google Android. Art and dalvik. https ://-source.android.com/devices/tech/dalvik/.

[3] Nikolaj Bjørner, Arie Gurfinkel, Ken McMillan, and Andrey Rybalchenko. Horn clause solvers for program verification. In *Fields of Logic and Computation II*, pages 24–51. Springer, 2015.

[4] Tool Interface Standard (TIS) Commitee. *Executable and Linking Format (ELF) Specification, Version 1.2*. Tool Interface Standard (TIS), 1995.

[5] Intel Corporation. Intel timeline : A history of innovation. https ://www.intel.com/content/www/us/en/history/historic-timeline.html.

[6] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual*. Intel Corporation, 2017.

[7] William F Dowling and Jean H Gallier. Linear-time algorithms for testing the satisfiability of propositional horn formulae. *The Journal of Logic Programming*, 1(3) :267–284, 1984.

[8] A Fog. Calling conventions for different c++ compilers and operating systems. *Technical University of Denmark.*, 2015.

[9] Inc. Free Software Foundation. Gcc, the gnu compiler collection, 2017. https ://gcc.gnu.org.

[10] Jean H Gallier. *Logic for computer science : foundations of automatic theorem proving*. Courier Dover Publications, 2015.

[11] Graeme Gange, Jorge A Navas, Peter Schachte, Harald Søndergaard, and Peter J Stuckey. Horn clauses as an intermediate representation for program analysis and transformation. *Theory and Practice of Logic Programming*, 15(4-5) :526–542, 2015.

[12] Alfred Horn. On sentences which are true of direct unions of algebras. *The Journal of Symbolic Logic*, 16(01) :14–21, 1951.

[13] P.A. Favier et A. Naim et F. Mesnard Jacques Tisseau, P. De Loor. Le langage prolog. 1990-2010.

[14] Jean-Marie Jacquet. Techniques d'intelligence artificielle. 2016-2017.

[15] Joxan Jaffar and J-L Lassez. Constraint logic programming. In *Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 111–119. ACM, 1987.

[16] Joxan Jaffar and Michael J Maher. Constraint logic programming : A survey. *The journal of logic programming*, 19 :503–581, 1994.

[17] Gérôme Laffineur. A declarative approach to simulation of the jvm and bytecode execution. 2017.

[18] Michaël Marcozzi. *Théorie des Langages de Programmation : Syntaxe et Sémantique : Eléments théoriques et exercices*. Presses universitaires de Namur, 1 2014.

[19] Frédéric Mesnard, Étienne Payet, and Wim Vanhoof. Towards a framework for algorithm recognition in binary code. In *Proceedings of the 18th International Symposium on Principles and Practice of Declarative Programming*, pages 202–213. ACM, 2016.

[20] nayuki. A fundamental introduction to x86 assembly programming, 2016. https ://www.nayuki.io.

[21] Roberto Paleari, Lorenzo Martignoni, Giampaolo Fresi Roglia, and Danilo Bruschi. N-version disassembly : differential testing of x86 disassemblers. In *Proceedings of the 19th international symposium on Software testing and analysis*, pages 265–274. ACM, 2010.

[22] Edward J Schwartz, JongHyup Lee, Maverick Woo, and David Brumley. Native x86 decompilation using semantics-preserving structural analysis and iterative control-flow structuring. In *Proceedings of the USENIX Security Symposium*, volume 16, 2013.

[23] Markus Triska. The Boolean constraint solver of SWI-Prolog : System description. In *FLOPS*, volume 9613 of *LNCS*, pages 45–61, 2016.

[24] Jan Wielemaker, Tom Schrijvers, Markus Triska, and Torbjörn Lager. Swi-prolog. *Theory and Practice of Logic Programming*, 12(1-2) :67–96, 2012.

[25] Wikipedia. Bitwise operation, 2017. https ://en.wikipedia.org.

[26] Gonzague Yernaux. Équivalence algorithmique par transformations de programmes logiques avec contraintes. 2017.

Annexe A

Désassemblage complet d'une section *.text*

```
Déassemblage de la section .text:

080482e0 <_start>:
 80482e0: 31 ed                 xor    ebp,ebp
 80482e2: 5e                    pop    esi
 80482e3: 89 e1                 mov    ecx,esp
 80482e5: 83 e4 f0              and    esp,0xfffffff0
 80482e8: 50                    push   eax
 80482e9: 54                    push   esp
 80482ea: 52                    push   edx
 80482eb: 68 60 84 04 08        push   0x8048460
 80482f0: 68 00 84 04 08        push   0x8048400
 80482f5: 51                    push   ecx
 80482f6: 56                    push   esi
 80482f7: 68 db 83 04 08        push   0x80483db
 80482fc: e8 bf ff ff ff        call   80482c0 <__libc_start_main@plt>
 8048301: f4                    hlt
 8048302: 66 90                 xchg   ax,ax
 8048304: 66 90                 xchg   ax,ax
 8048306: 66 90                 xchg   ax,ax
 8048308: 66 90                 xchg   ax,ax
 804830a: 66 90                 xchg   ax,ax
 804830c: 66 90                 xchg   ax,ax
 804830e: 66 90                 xchg   ax,ax

08048310 <__x86.get_pc_thunk.bx>:
 8048310: 8b 1c 24              mov    ebx,DWORD PTR [esp]
 8048313: c3                    ret
 8048314: 66 90                 xchg   ax,ax
 8048316: 66 90                 xchg   ax,ax
 8048318: 66 90                 xchg   ax,ax
 804831a: 66 90                 xchg   ax,ax
```

```
 804831c: 66 90                 xchg   ax,ax
 804831e: 66 90                 xchg   ax,ax

08048320 <deregister_tm_clones>:
 8048320: b8 1f a0 04 08        mov    eax,0x804a01f
 8048325: 2d 1c a0 04 08        sub    eax,0x804a01c
 804832a: 83 f8 06              cmp    eax,0x6
 804832d: 76 1a                 jbe    8048349 <deregister_tm_clones+0x29>
 804832f: b8 00 00 00 00        mov    eax,0x0
 8048334: 85 c0                 test   eax,eax
 8048336: 74 11                 je     8048349 <deregister_tm_clones+0x29>
 8048338: 55                    push   ebp
 8048339: 89 e5                 mov    ebp,esp
 804833b: 83 ec 14              sub    esp,0x14
 804833e: 68 1c a0 04 08        push   0x804a01c
 8048343: ff d0                 call   eax
 8048345: 83 c4 10              add    esp,0x10
 8048348: c9                    leave
 8048349: f3 c3                 repz ret
 804834b: 90                    nop
 804834c: 8d 74 26 00           lea    esi,[esi+eiz*1+0x0]

08048350 <register_tm_clones>:
 8048350: b8 1c a0 04 08        mov    eax,0x804a01c
 8048355: 2d 1c a0 04 08        sub    eax,0x804a01c
 804835a: c1 f8 02              sar    eax,0x2
 804835d: 89 c2                 mov    edx,eax
 804835f: c1 ea 1f              shr    edx,0x1f
 8048362: 01 d0                 add    eax,edx
 8048364: d1 f8                 sar    eax,1
 8048366: 74 1b                 je     8048383 <register_tm_clones+0x33>
 8048368: ba 00 00 00 00        mov    edx,0x0
 804836d: 85 d2                 test   edx,edx
 804836f: 74 12                 je     8048383 <register_tm_clones+0x33>
 8048371: 55                    push   ebp
 8048372: 89 e5                 mov    ebp,esp
 8048374: 83 ec 10              sub    esp,0x10
 8048377: 50                    push   eax
 8048378: 68 1c a0 04 08        push   0x804a01c
 804837d: ff d2                 call   edx
 804837f: 83 c4 10              add    esp,0x10
 8048382: c9                    leave
 8048383: f3 c3                 repz ret
 8048385: 8d 74 26 00           lea    esi,[esi+eiz*1+0x0]
 8048389: 8d bc 27 00 00 00 00  lea    edi,[edi+eiz*1+0x0]

08048390 <__do_global_dtors_aux>:
 8048390: 80 3d 1c a0 04 08 00  cmp    BYTE PTR ds:0x804a01c,0x0
 8048397: 75 13                 jne    80483ac <__do_global_dtors_aux+0x1c>
 8048399: 55                    push   ebp
```

```
 804839a: 89 e5                 mov    ebp,esp
 804839c: 83 ec 08              sub    esp,0x8
 804839f: e8 7c ff ff ff        call   8048320 <deregister_tm_clones>
 80483a4: c6 05 1c a0 04 08 01  mov    BYTE PTR ds:0x804a01c,0x1
 80483ab: c9                    leave
 80483ac: f3 c3                 repz ret
 80483ae: 66 90                 xchg   ax,ax

080483b0 <frame_dummy>:
 80483b0: b8 10 9f 04 08        mov    eax,0x8049f10
 80483b5: 8b 10                 mov    edx,DWORD PTR [eax]
 80483b7: 85 d2                 test   edx,edx
 80483b9: 75 05                 jne    80483c0 <frame_dummy+0x10>
 80483bb: eb 93                 jmp    8048350 <register_tm_clones>
 80483bd: 8d 76 00              lea    esi,[esi+0x0]
 80483c0: ba 00 00 00 00        mov    edx,0x0
 80483c5: 85 d2                 test   edx,edx
 80483c7: 74 f2                 je     80483bb <frame_dummy+0xb>
 80483c9: 55                    push   ebp
 80483ca: 89 e5                 mov    ebp,esp
 80483cc: 83 ec 14              sub    esp,0x14
 80483cf: 50                    push   eax
 80483d0: ff d2                 call   edx
 80483d2: 83 c4 10              add    esp,0x10
 80483d5: c9                    leave
 80483d6: e9 75 ff ff ff        jmp    8048350 <register_tm_clones>

080483db <main>:
 80483db: 55                    push   ebp
 80483dc: 89 e5                 mov    ebp,esp
 80483de: c7 05 20 a0 04 08 05  mov    DWORD PTR ds:0x804a020,0x5
 80483e5: 00 00 00
 80483e8: a1 18 a0 04 08        mov    eax,ds:0x804a018
 80483ed: ba 20 00 00 00        mov    edx,0x20
 80483f2: 01 c2                 add    edx,eax
 80483f4: a1 20 a0 04 08        mov    eax,ds:0x804a020
 80483f9: 0f af c2              imul   eax,edx
 80483fc: 5d                    pop    ebp
 80483fd: c3                    ret
 80483fe: 66 90                 xchg   ax,ax

08048400 <__libc_csu_init>:
 8048400: 55                    push   ebp
 8048401: 57                    push   edi
 8048402: 56                    push   esi
 8048403: 53                    push   ebx
 8048404: e8 07 ff ff ff        call   8048310 <__x86.get_pc_thunk.bx>
 8048409: 81 c3 f7 1b 00 00     add    ebx,0x1bf7
 804840f: 83 ec 0c              sub    esp,0xc
 8048412: 8b 6c 24 20           mov    ebp,DWORD PTR [esp+0x20]
```

```
 8048416: 8d b3 0c ff ff ff     lea    esi,[ebx-0xf4]
 804841c: e8 6b fe ff ff        call   804828c <_init>
 8048421: 8d 83 08 ff ff ff     lea    eax,[ebx-0xf8]
 8048427: 29 c6                 sub    esi,eax
 8048429: c1 fe 02              sar    esi,0x2
 804842c: 85 f6                 test   esi,esi
 804842e: 74 25                 je     8048455 <__libc_csu_init+0x55>
 8048430: 31 ff                 xor    edi,edi
 8048432: 8d b6 00 00 00 00     lea    esi,[esi+0x0]
 8048438: 83 ec 04              sub    esp,0x4
 804843b: ff 74 24 2c           push   DWORD PTR [esp+0x2c]
 804843f: ff 74 24 2c           push   DWORD PTR [esp+0x2c]
 8048443: 55                    push   ebp
 8048444: ff 94 bb 08 ff ff ff  call   DWORD PTR [ebx+edi*4-0xf8]
 804844b: 83 c7 01              add    edi,0x1
 804844e: 83 c4 10              add    esp,0x10
 8048451: 39 f7                 cmp    edi,esi
 8048453: 75 e3                 jne    8048438 <__libc_csu_init+0x38>
 8048455: 83 c4 0c              add    esp,0xc
 8048458: 5b                    pop    ebx
 8048459: 5e                    pop    esi
 804845a: 5f                    pop    edi
 804845b: 5d                    pop    ebp
 804845c: c3                    ret
 804845d: 8d 76 00              lea    esi,[esi+0x0]

08048460 <__libc_csu_fini>:
 8048460: f3 c3                 repz ret
```